Introduction

Do you want to learn how to program in Pascal? Well, you've come to the right place. Here, we'll try to pass on to you the different skills needed to be a Pascal programmer, from the bare basics to the advanced techniques, this page is all you need.

About This WebPage

This page is an entire reference of all the possible things you could ever, or never, want to learn about Pascal. There is both an interactive tutorial to guide thoughout your programming path and an **A**lmost **C**omplete **R**eference to all the terms ever related to Pascal.

How to use this WebPage

Well, you could either:

1. Use the tutorials to guide you through the world of Pascal, or
2. Read through the whole **A**lmost **C**omplete **R**eference and end up going crazy.

We *highly* recommend that you take the first option as it is much easier, (and less boring) than the second option. (After all, the ACR was designed to be a reference, not a manual...)

When examples are given, they'll be presented in a "Type", "Run", "Analyse" system, clearly marked by these three distinctive bars:

TYPE

```
program This_does_nuthin;

  uses
    Crt;

  begin
    { Be sure to type this the same way you see it. }
    Writeln('Wabba, wabba!');
  end.
```

RUN

```
Wabba, wabba!
```

ANALYSE

The program above is the simplest example of using both comments and the Writeln statement. Ha! How's that for style, eh? This is fantastic! Long live Pascal! Long live Pascal! Notice the elegance in style! The beauty in writing! The wonder of language! How marvelous Pascal is! Marvelous, just marvelous!

As you can see from the example above, **Type** is the program which you will type, **Run** displays the output of the program, and **Analyse** gives an analysis of the program. This will be how all example programs are introduced.

As far as possible, references back to the manual have been avoided. Every time something new is introduced, we will try to give you an accurate, yet simple explanation without having you refer back to the manual.

Recommended Software & Hardware

We recommend the following programs for the best programming environment:

- Windows 95 or NT
- Turbo Pascal 7.0, and
- Borland Delphi 2.0

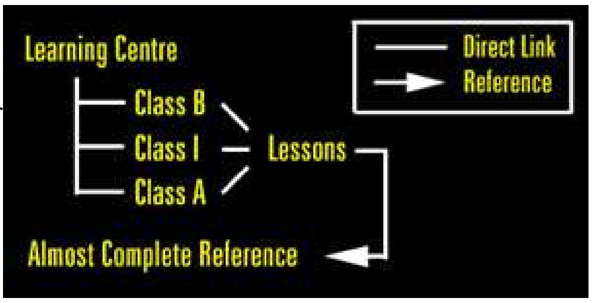
If you don't have Turbo Pascal or Delphi 2.0, you can find out how to get them at www.borland.com, or try to get other Pascal compilers on the market.
We also recommend the following hardware for the best programming environment:

- a 80486 processor or above
- 8MB of RAM or more
- 20MB of free hard disk space or more
- at least a 14.4 kps modem

You don't really need to satisfy the requirements to use this page efficiently. Remember, this is the *best* setup, and you don't neccesarily need the best.

How this page is organized

All the lessons start from the Learning Centre, where they are branched into three groups, B, I, and A. B for Beginners, I for Intermediates, and A for Advanced.
When needed, these lessons might refer to the Almost Complete Reference for definitions of new terms and such. Select Back from your browser to continue your lesson at where you left. You can always go back to read the definition again later.
Remember, the lessons get progressively more and more complex, so we recommend that you follow the lessons in the order they were placed. This way, you won't get confused that easily.



Icons Used In This Page

These icons are used to flag important or useful information:

Tips - Useful information you might need.

Ideas - Ideas for creating better programs.

Rem - A little reminder, just in case.

Stop! - **Stop!**, before something goes wrong!

URL - Provides a link to another related topic.

Now where?

First of all, you can start your lessons immediately by selecting **LC** below, but if you want to explore around, go ahead. Remember, the pace, the difficulty, everything, is decided by you. So, go, and enjoy the great and mighty world of Pascal programming by your own. Relax, have fun, and enjoy yourself. Bon Voyage!

Learning Centre

CLASS B CLASS I CLASS A

Welcome to the Learning Centre! This is where all your lessons start, and they are arranged in this order: Beginners, Intermediates, and Advanced. We call them "Classes", and they're labelled B, I, and A. Here's a brief discussion of what each "Class" does.

Class B - This is where you start your lessons as a new Pascal programmer! We'll introduce to you the skills, the techniques, and the ways to program well and how to use Turbo Pascal effectively. Everything you'll ever need is here to get started on programming!

Class I - After you're finished with Class B, you can now proceed with Class I, the intermediate class. Here, things get thicker, but still easy enough to understand. Remember, you can also come in here immediately if you believe you already got the basics right. You're perfectly welcome!

Class A - This is where your advanced lessons begin. After getting a good grasp at the topics in Class B & I, the next step you should take is to come here and learn more! Expect to find things a wee bit more difficult to understand, but trust me, just think for a while, and you'll get it. Good luck!

Class B

Here are the tutorials for Class B in chronological order, each accompanied with a short description about itself. You can also download the code samples for all the example programs! Now all you have to do is to just select one of these tutorials, and start your self-paced Pascal course.

***Day 1* - Introduction to Programming**
This beginning tutorial will teach to you the complete basics of programming, what are variables, constants, and whatnot. After this, you can finally begin to understand what programming is, and what's in it - Variables, Control Flow, and so on.

***Day 2* - Elements of Pascal**
In this tutorial, you will be introduced to the different parts of a Pascal program and what each one of them does. It includes a 'dissection' of a Pascal program, a walkthrough of what each piece does, and a final conclusion to round up all the points.

***Day 3* - Variables and Constants**
Here, we introduce to you to variables and constants, two very similar, but different objects. We explain about the concept of variables, and operations we can do with them. After which, we will talk about constants, how they work, and why we need them in the first place.

***Day 4* - Control Flow in Pascal**
This tutorial will teach you the many different control flow loops there are in Pascal. Without these, your programs will be one-dimensional! Each control flow loop is accompanied with an example program. With these loops, you can do almost anything you want!

***Day 5* - Procedures and Functions**
You can't claim to be a structured programmer if you don't know Procedures and Functions, so here they are. Explaining scope of variables, parameters, and such, this is the perfect introduction to structured programming, without even knowing what it means!

***Day 6* - Comments and Other Neat Stuff**
Your last day is nothing but a bunch of miscellaneous tips strung together, but it's definitely indispensible. From formatting Writeln to getting input from the user, this day does it all... and more! A perfect end to move on to Class I. (Note: Follow the instructions at the end!)

Class B
Day 1 - Introduction to Programming

What you are going to learn today:

- What programming is all about
- The different parts of programming
- And how all this relates to Pascal

Well, you're about to step into the wild, wacky, and completely out-of-control world of Pascal programming. But before you do so, you need to know the bare-bone basics of programming, and this is what this tutorial tries to do. This tutorial will introduce you to programming, what it is, how it works, and ways to make programming work for you, not you working for the program.

What programming is all about

Well, programming is about *controlling the computer*. That's why it's called programming. And how do you program? First, you use a language. Then you get a compiler, then Voila! You're a programmer! Pascal is an example of a programming language, and the only compilers on the market for Pascal come from Borland. Visit their website to find out how to get Turbo Pascal 7.0, if you don't have it yet. That's the compiler we'll be using throughout this webpage.

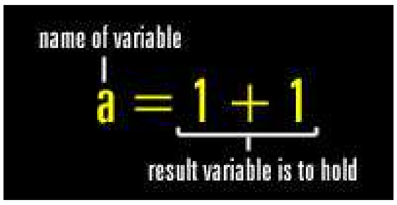
URL - The Borland site is at www.borland.com

Remember, programming can be both a fun and painful hobby, depending on how you view it. So, look on the bright side, and take it lightly. You're dealing with computers, remember? So just sit back and enjoy the ride!

The different parts of programming

Now, even before we even get into Pascal, let's ask the question you've all been waiting for: What is inside programming? Well, to answer this question, we'll try to tell you the typical parts of every programming language, whether it's Pascal, C++, or BASIC. Of course this list won't be complete, or accurate, but for the moment we'll be using this list to guide us on. Because this is pretty complex stuff, we recommend that you actually *skip this entire list* and go on to Day 2. Don't worry, everything will be explained in greater detail in the later tutorials.

Variables - storing values


name of variable
$\underset{\text{result variable is to hold}}{a = \underbrace{1 + 1}}$
result variable is to hold

Well, let's start off with *variables*. (Devilish creatures, certainly) Remember those Algebra classes? Well, don't worry if you don't, because we're going to put you on a crash course right through!

All programming language work on something called *variables*, which work the same way algebra does. Each *variable* has a name and contains a value. For example, a *variable* can be named *a*, and contains the value *2*. Thus you can say that a + a = 4 ! Easy, no?

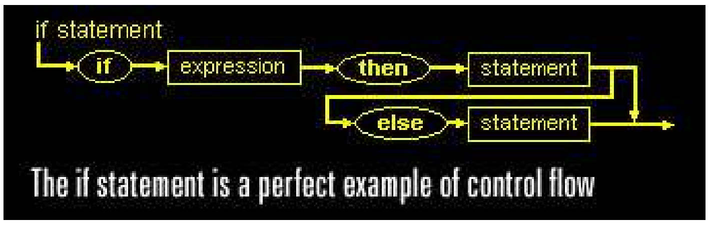
Now, how do programming languages handle variables? Well, first, they put the variable they want to change in front, then the value it is to hold behind. Okay, so it seems weird, but its better than making a whole new concept for us to learn right?

Constants - staying the same

Essentially, constants are what they are, constant. A constant just simply replaces something else, like how some publications replace certain words with asterisks when nessecary. To give an example, let's say that we replace every appearance of the word "lamb" with "head" to make the sentence "Mary had a little lamb" to become "Mary had a little head". Thus, you could say that the word "head" was a constant representing "lamb", with apologies to all lamb-lovers out there of course.

Control Flow - changing directions

When you run through a list of instructions, you normally go through them step by step, but what if that's not the case? What if



you are to press the red button when a tone is heard and the green one when not? What if you are supposed to scream when a meltdown occurs but shout when a fire starts? That's where control flow comes in to change the course of the program. Control flow is nessecary because you can't expect everything to be like a set of instructions. Changes have to be made.
For example, you can say, "Jump off the building *if* it is on fire, *else* continue typing your report." The statement above accomplishes something a list just can't do - change the direction.

Procedures - sub-routines

When you need to do something again and again, or to process some information, the programming mechanism you use is called a procedure. A procedure is something like a fixed set of instructions which can be repeated again and again. For example, a procedure may state, "Get the number of customers in the room and get the appropiate number of chairs". Thus, this procedure can be used in a concert, or a party, or any one of those lavish social occasions people never really care to go to. That's how a procedure works.

Functions - sub-routines which return values

A function is just like a procedure, except that it returns a value. For instance, we may have a procedure that accepts two numbers, one representing the number of Godzillas there are in a room, the other representing the number of innocent bystanders passing by, and calculates the average number of screams every 5 seconds. This is called a function because it returns a value, which is the average number of screams.

Comments - little notes

The final thing we have on our list are comments, which allow the programmer to add notes to his code. This makes the code easier for other people to read, and easier to change when debugging.

How this all fits into Pascal

Well, you'll see very soon that Pascal implements all these little aspects of programming in its own way, and in its own style too. For now, go get a can of Coke from the fridge. You earned it. Cheers! Now, go on to Day 2, where you *really* ***start programming!***

Class B
Day 2 - Elements of Pascal

What you are going to learn today:

- How do you start up Turbo Pascal
- Your first Pascal program
- And an analysis of each element in your program

Today, you're going to write your first Pascal program, and you're going to analyse it. No, forget those images of Biology class, you're just going to look through the dissection part briefly.

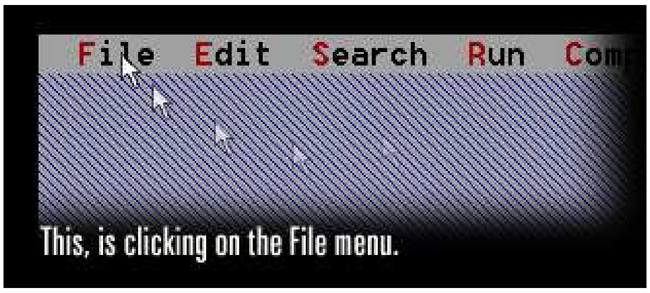
Starting Up Turbo Pascal

Ahhhh, this is easy. First, I presume you've already installed Turbo Pascal, right? Now just follow these easy to remember directions:

1. Go to the directory where you installed Turbo Pascal (e.g. **cd tp**)
2. Go to the **bin** subdirectory (e.g. **cd bin**)
3. Type **turbo** to start up Turbo Pascal

In case you're wondering about what **bin** means, well, it means that whatever goes there should be consigned to the bin, or maybe some other form of mild destruction. Well, actually, that was just a joke. The bin directory actually means *binary*, and you shouldn't fool around with binary stuff, right? That directory contains all the program files, and you could really foul up there.
Now, just wait for a moment, and Turbo Pascal would appear with all its glory on your screen, filling it up with its large, grey, empty desktop. If this is its first startup, or you had quit it with an editor window open the last time you ran it, a new window would appear on your screen.
Well, what if it doesn't? Then just follow the instructions, just follow the instructions...

1. Go to the **File** menu. (Click on the word "File")
2. Select New from the pop-up menu that appears



This, is clicking on the File menu.

Poof! A brand new window appears for you. Now, you can go on to do your magic, to type your very first program in Pascal...

Your very first Pascal program

The program you're about to type is only one method to implement the many techniques in programming. The user is invited to make his or her own changes to the program code to fit the situation. So here goes.

TYPE

```
program FirstProg;

  begin
    { I always wanted to do this. }
    Writeln('Hel-lo nurse... I mean, World!');
  end.
```

RUN

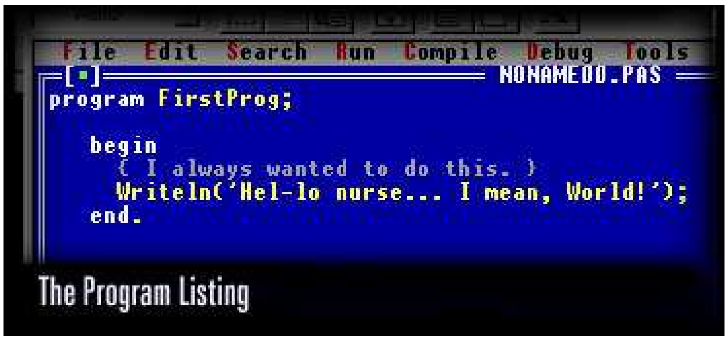
(To run, go to the Run menu and select *Run*. To see the output, go to the Debug menu and select *User Screen*. Don't worry, we'll explain it later.)

```
Hel-lo nurse... I mean, World!
```

ANALYSE

Now, that wasn't too difficult! Let's analyse the program line by line.

- ***Line 1*** *- this is just a line declaring the program to be named FirstProg. In case you're wondering, this line is for humans to read, not for the compiler. See the semicolon after the name FirstProg? It's very special. You'll see why later.*
- ***Line 3*** *- this tells the compiler that you're beginning a block of code for it to read. The real code comes next.*
- ***Line 4*** *- this line is in curly brackets, signalling to the compiler that this is just a comment and should be ignored.*
- ***Line 5*** *- Now, this is the line that makes the magic. Catch the command at the beginning, Writeln? After that, comes a bracket and the line that gets printed follows behind with a closing bracket. See the semicolon again? We'll explain that very soon...*
- ***Line 6*** *- this tells the compiler that it is the end of the code block and it can stop reading code now. Notice that the semicolon is replaced by a full stop? Ahhh, food for thought.*



```
program FirstProg;

  begin
    { I always wanted to do this. }
    Writeln('Hel-lo nurse... I mean, World!');
  end.
```

The Program Listing

Now, let's see how this Pascal program *really* works. After Line 3 which starts the block of code, we have Line 4, which is a comment, telling us that "I always wanted to do this". The compiler ignores this line completely, giving us a fighting chance to make our code readable for other humans. In fact, that's exactly what comments are used for. Neat, eh? Let's go on. Now, in line 5, we have the line that does all the magic. But how does it work? Well, we used the Writeln command, and we passed it the string we wanted it to print. Through some technical mumbo-jumbo and a lot of praying, the string gets its way onto the screen. Isn't that nice? Well, I bet you know what line 6 is for then. Yes, it's for ending off the program. Right on.

But what about the semicolon? Well, here's how it works. Pascal states that you end every statement or any block of code with a semicolon. So, line 1, which is a statement, ends with a semicolon. Line 5 too. All the statements in this program end with a semicolon.
Then what about the last line? Well, you can say that a Pascal program is like a long, unwieldly sentence. You must of course end off with a full stop. So, it's the same with the very last block of code. You must put a full stop instead of a semicolon because it is the end of the program. Logical, isn't it? End Of Day 2 Finito. That's it. Day 2 is done and over with. Now, get ready for Day 3, which marks the beginning of your marathon Pascal knowledge race. Ready? Well, take your time... This is meant to be a tutorial, remember? So get up. Day 3 is waiting for you.

Class B
Day 3 - Variables and Constants

What you are going to learn today:

- What variables and constants are
- Some examples of variables and constants
- How variables are implemented in Pascal
- And an variables example program

So, you've typed your first program, dissected it line by line, and now you're ready for anything. Well, today you're going to learn about variables and constants, nothing particularily difficult, so just sit back and enjoy!

What are Variables and Constants?

That's an interesting question. How would you define a variable? Well, ever learnt Algebra before? It's the same concept. Just take a name, and assign it a value. It's *that* simple. For instance, you could say that a = 2. So, a + a is the same as 2 + 2, which equals 4. Easy does it.

Well, then you may ask, "If it's so darn simple, why call them variables, and not an algebric number or something?" Ahh, good question! You see, the name comes from the very nature of variables in programming. They can *vary*. Sure, I know it wasn't like that in Algebra class, but our aim now is not to find out what *x* is, but to instead use *x* as just a storing place for a value. Get the idea? Our *x* is just a storing place for 2.

Then what about constants? Well, constants can be said to be the exact opposite as variables. Instead of just being storage space, the aim of a constant is to replace values. For example, if we had replaced every occurance of the word "lamb" with the word "head" in a certain famous nursery rhyme, we would had got:

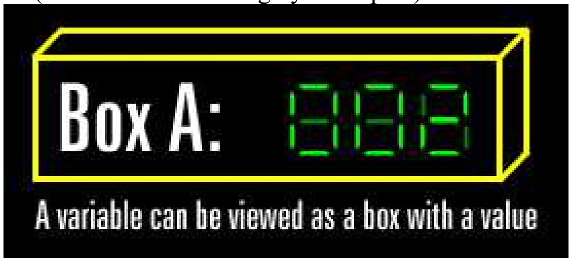
"Mary had a little head,
its fleece was white as snow.
And everywhere that Mary went,
her head was sure to go."

Yes, it's a very *strange* version of the original classic, but that's only if you read it literally. If I had said that "head" is just a constant representing "lamb", it would have made perfect sense. Never thought of it that way, eh? Well, better start thinking about it. Constants were created to make program code more readable by replacing meaningless numbers with a meaningful name. A very nifty feature!

Some more examples of Variables and Constants

Still can't get the idea? Don't worry, here are a few more examples to help you out. (We call this 'Learning by Example'.)

One method we can use for explaining a variable is to use the analogy of a box. Imagine that a variable is actually a box carrying the value which it stores. Now, you can actually take the value out, and put in a new one. After that, you can look up the value in the box and do things with it. You could also copy the value into other boxes, or take the values from other boxes and put them in your box. Variables are *very* flexible. They can store values from almost any other source you specify.



But here's where the concept of data types come in. You see, just like in real life, boxes come in all shapes and sizes. So, some boxes can't store certain values beyond their limits. Ahh, you guessed there was a catch! The different data types will be introduced to you later. For now, however, let's just talk about constants first.

Constants can be compared to a look-up table, something you'll use to look up unknown terms. (This is similar to the use of the word 'software' in license agreements, if you ever read them.) When you come across this 'unknown term', you will have to look up this table and understand what it means. For instance, if you declare a constant 'Name' representing "Tan Chun Ghee", the sentence "I think Name is really great!" would translate to "I think *Tan Chun Ghee* is really great!" This would be the same with the sentence "Name is my idol!" which would translate to "*Tan Chun Ghee* is my idol!" As you can see, "Name" is explicitly replaced with the text that it represents.

How Variables and Constants are implemented in Pascal

Yes, now that you have understood the underlying concepts behind variables and constants, you can finally find out how to implement them in Pascal!

Implementing Variables

Variables in Pascal are declared using the 'var' keyword. You need to *declare* a variable before you can use it. Why? Well, it's like if you were making the box for the value to be stored. Then, after the box is finished, you can start placing the value into it. You can't use a box if you don't make one! These are how declarations look like:

```
var
  Number, Value: integer;
  Text: string;
  TrueOrFalse: boolean;
```

The syntax for a declaration is to put the *name of the variable*, a colon, the *data type* of the variable, and finally, the grand semicolon. You can also declare more than one variable in one line by separating them with commas. Now that reminds me, we still haven't told you the different data types, haven't we? Well, here they are, shamelessly lifted from the Turbo Pascal help:

Type	Range	Precision
Shortint	-128..127	Whole numbers only
Integer	-32768..32767	Whole numbers only
Longint	-2147483648..2147483647	Whole numbers only
Byte	0..255	Whole numbers only
Word	0..65535	Whole numbers only
Boolean	True or False	Whole numbers only
Real	2.9e-39..1.7e38	11-12
Single	1.5e-45..3.4e38	7-8
Double	5.0e-324..1.7e308	15-16
Char	a character	Characters only
String	255 delightful characters!	Text only

Of course, those are not all the data types you can have in Pascal. In fact, you can even make your own data types, so there really isn't any limit! Variables can be any size you want them to be.

Implementing Constants

In Pascal, constants are listed under the 'const' keyword, just like the look-up table concept explained above. A typical list of constants would look like this:

```
const
  Pi = 3.14;
  Gravity = 9.8;
  Golden = 1.6;
```

Because these numbers are meaningless by their own, we give them more meaningful names to make our code more readable. Imagine reading an entire maths textbook which refers to pi by its value! Unthinkable!

Below is the example program listing for today. Open a new code window to type in the code. Ahh, the three famous bars!

TYPE

```
program VariableMania;

  const
    Interest = 2;

  var
    Money, Debt: integer;

  begin
    { This is a very sad story of a man and a bank. }
    { Moral: Never tell a story about a man and a bank. }

    Debt := 100;
    Writeln('There was once a man who owned a bank $', Debt, '.');

    Money := 99;
    Writeln('However, he only had $', Money, '.');

    Writeln('After one year, an interest of ', Interest, '% was issued.');
    Debt := Debt + (Debt div 100 * Interest);
    Writeln('His debt increased by $', (Debt div 100 * Interest), '.');
    Writeln('Now, his debt was $', Debt, '.');
  end.
```

RUN

```
There was once a man who owned a bank $100.
However, he only had $99.
After one year, an interest of 2% was issued.
His debt increased by $2.
Now, his debt was $102.
```

ANALYSE

Once again, let's have a line-by-line analysis of this program. It looks long, but inside it's simpler than you think!

- ***Line 4*** *- This is a declaration for the constant 'Interest'. Now, whenever the compiler sees 'Interest', it'll return this number instead.*
- ***Line 7*** *- Here are the declarations of variables 'Money' and 'Debt'. Both are* ***integers****, which only allow them to store whole numbers, but as you'll see later, this arrangement makes the program much easier to read.*
- ***Line 13*** *- This is where the program really starts. See the* ***:=*** *sign? Whenever that sign is used, it means that the value to the* ***left*** *of the sign will now store the value to the* ***right*** *of the sign. So in this case, 'Debt' now stores the value 100.*
- ***Line 14*** *- This line prints out a message that changes with the value of 'Debt'. If you inspect the line closer, you'll see that after the first string, there is a comma, the variable name, and another comma, followed by the next string. When this line gets printed, The value of the variable is printed onto the screen.*
- ***Line 16 & 17*** *- These lines do exactly the same thing as lines 13 and 14, except that they print out 'Money' instead of 'Debt'.*
- ***Line 19*** *- This line uses a constant to print out the sentence. The compiler replaces the word 'Interest' with the number 2. Once again, this uses the Writeln command to output the sentence.*
- ***Line 20*** *- Here, a calculation is carried out* ***at the right*** *of the* ***:=****, and the value is stored into 'Debt'. What it does into to take the current value of 'Debt', add it to Debt divided by 100 times 2, and place the new value into 'Debt'. In this calculation, division is replaced by 'div', and multiplication is replaced by '*'. You'll see why later.*
- ***Line 21 & 22*** *- These lines output the amount of interest and the new value for 'Debt'.*

Remember the procedure to run programs in Day 1? Well, here's how it works: When you first ran the program, the output flashes by so quick that it's nothing but a flicker on your screen. When you select 'User Screen', or press Alt-F5, you are just simply seeing what the user would see after he exits the program.
Now, had fun writing that? You might notice that our mathematical operations seemed to have changed a bit. Well, the standard for the four mathematical operations are +, -, *, and, /, with the slash meaning division, but you may notice that in our program, the slash is replaced by the keyword 'div'. This is because we're dealing with integers here, and the / symbol only returns real numbers. So we calculate integers with the 'div' keyword instead. Look up the help for more details.

End of Day 3

All right! It's over for now. Get up and give yourself a little *stretch*. You've just gone through a crash course through Variables and Constants, and now we're going on to Day 4, where we talk about "Control Flow", so hang on! It's not over yet!

Class B
Day 4 - Control Flow in Pascal

What you are going to learn today:

- What control flow is
- The different types of control flow loops
- And example programs to demonstrate

Control Flow is the control of how a program executes in Pascal. It is neccesary if you don't want the program code to run sequentially or when you need to run code only under certain conditions. This tutorial will teach you the different control flow loops.

So, what is control flow?

Well, just imagine that you're Bill Gates. You're overflowing with money, and you're planning to get a can of Coke from the Coke machine right outside your office. You step out, take a deep breath, and dig out your *heavy* wallet from your pocket. Ahh, there are two possibilities for what that will happen after this:

1. Realising that your wallet only has notes in it, you groan and mutter something about "the advancement of mordern technology", then go off looking for the nearest human Coke vendor...

2. Or, you take out all the coins you have and dump them one by one into the Coke machine, hoping that Coke isn't sold out. *Yet*.

Well, congratulations! You've just gone through your first course through control flow! When you were standing in front of the Coke machine, there were two possibilities, right? *If* you didn't have coins, you would have to go to a human vendor. *Otherwise* you would buy your Coke as normal. Well, that's control flow! Control flow is simply just *controlling* how the program *flows*, making decisions, looping code, and other things like that. Whenever you need to have the program do things that aren't sequential, you'll have to use control flow to modify the program sequence. Otherwise, it just can't be done. In Pascal, there are a few control flow loops for you to use. Most of them are pretty logical, so without further ado, let's go on to find out what they are.

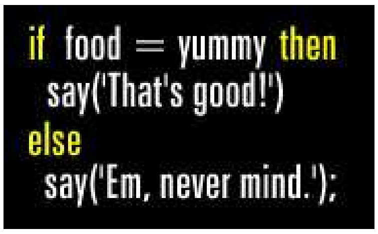
The different control flow loops follow...

Here they are, in order of usefulness, the grand, the fantastic, the wonderful, and the unbelievably complex control flow loops! Here for your personal reading enjoyment!

1. The if-else statement

When you need to make decisions in code, the if-else statement fits the job perfectly. A typical if-else statement looks like this:

```
  if a = 30 then
    Writeln('A is equal to 30')
  else
    Writeln('A is not equal to 30');
```



```
if food = yummy then
  say('That's good!')
else
  say('Em, never mind.');
```

An if-else statement is actually a conditional statment that will execute code only if a condition is true. If the 'else' keyword is there and the condition is not true, then the second line of code will be executed instead. The condition is stated between the keywords 'if' and 'then'. In this case, the condition is 'a = 30' which tests whether the value of a is equivalent to 30.

Rem - The '=' is used for testing for equality, but the ':=' is for assigning values. Don't mix them up! Use the '=' for testing if two values are the same, but ':=' if you want to assign something a value.

if a is equivalent to 30 the first line code would execute, printing "A is equal to 30" on the screen. However if a is not equal to 30, it would execute the line immediately after 'else', printing "A is not equal to 30". Magic! Pure magic!

Now, you may ask two questions. One, how come the first line of code has no semicolon at the end? Simple. There's no semicolon as the entire if-else thing is one big statement! If the 'else' part is removed, then that line would require a semicolon. In contrast...

```
  if a = 30 then
    Writeln('A is equal to 30');
```

If a is equal to 30 *then* the program will print "A is equal to 30". Otherwise, it won't do anything.

The second question that might be in your mind would be how do you execute more than one line in the if-else statement? Logically, you would do this:

```
  if a = 30 then
    Writeln('Congrats!')
    Writeln('A is equal to 30')
  else
    Writeln('A is not equal to 30');
```

However, this is **totally wrong**! Whenever you need to type more than one line in a place where only one line is accepted, you need to use a new code block! Yes! You actually type another set of 'begin' and 'end'! Thus, what the above should look like is this:

```
  if a = 30 then
    begin
      Writeln('Congrats!');
      Writeln('A is equal to 30');
    end
  else
    Writeln('A is not equal to 30');
```

Notice that there's also no semicolon after the 'end'. Pascal treats the whole code block as one line of code! Well, confused? Ahh, there's nothing better than an example program to solidify everything...

TYPE

```
program TestTheNumber;

  var
    a: integer;

  begin
    a := 12;

    { This tests if a is less than 50 }
    if a < 50 then
      Writeln('A is less than 50')
    else
      Writeln('A is more then 50');
  end.
```

RUN

```
A is less than 50
```

ANALYSE

Dissection Time! This program is easier than you think! You may see some unfamiliar signs, but besides that, it's all logic...

- ***Line 4*** *- We declare the variable a here.*
- ***Line 7*** *- Now, we assign the value 12 into a.*
- ***Line 10*** *- Here, we test if a is* ***less than*** *50. That < sign means 'less than'. Other operational and relational operators will be introduced to you later. If a is less then 50 then the code on line 11 will execute, else the code on line 13 will execute.*
- ***Line 11*** *- This line executes only if a is less than 50*
- ***Line 13*** *- This line executes only if a is not less than 50*

Ahh, you learnt something new today. The mathematical sign for less than works the same way in Pascal! You may be wondering, what other operators are there? Well, here they are:

Operator	Operation
=	Equal
<>	Not equal
<	Less than
>	More than
<=	Less or equal
>=	More or equal

Also, you can have multiple tests in one condition by enclosing them in brackets and linking them with 'and', 'or', or 'xor'. For example:

```
  if (a > 50) and (a < 100) then
    Writeln('A is more than 50 and less than 100');
```

Only when the first condition *and* second conditions are met, will the message be printed out. Wondering what the other two operators do? Here's something called a truth table to help you:

		and	or	xor
True	**True**	True	True	False
True	**False**	False	True	True
False	**True**	False	True	True
False	**False**	False	False	False

As you can see, 'and' only returns true if all conditions are true, 'or' returns true as long as one of them is true, and finally, 'xor' returns true only if one is true and the other false. Don't worry if you can't get it for now, it's really simple once you get the hang of it.

2. The for loop

When you need to repeat a certain bit of code again and again, the <u>for</u> loop is just what you need. It lets you specify the exact number of times the code is repeated, giving you the flexibility you always dreamed of. (Then again, who dreams about making code repeat as many times as you want?)

The for loop uses a counter which increases by one each time the code is repeated. That means that you need to declare a variable before you can use the for loop. The for loop looks like this:

```
  for i := 1 to 10 do
    Writeln('Help! I'm being multiplied!');
```

Ah, the beauty of it all. As you can see, after the word 'for', we place the name of the counter variable. Then, we ask it to loop from 1 to 10, successfully repeating the line of code 10 times. Also, every time a loop is finished, i increases by 1. And yes, you can actually use i in your code! Look at the example below:

```
  for i := 1 to 10 do
    x := x + i;
```

Assuming that x starts of with 0, the end result would be 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10, which equals 55. Of course there are other uses for this feature, but this exercise will left up to the reader.

Tips - A quick way to calculate the above is to use the Gauss Theorem, which works because 10+1, 9+2, 8+3, 7+4, 6+5, all equal to 11. Thus, the sum above is actually equivalent to 5 times 11. Go figure that out for yourself...

Now you may ask, how do you make the counter go backwards? Well instead of typing 'to' type 'downto'. The counter will then count backwards. See?

```
  for i := 10 downto 1 do
  x := x + i;
```

Now that you know everything there is to know about the for loop, let us once again have another example program to round the for loop up. (After all, the fun part is in the typing, isn't it?)

TYPE

```
program LoopyLoop;

  var
    i: integer;

  begin
    for i := 1 to 5 do
      Writeln('This is loop number ', i, '.');
    for i := 5 downto 1 do
      Writeln('This is loop number ', i, '.');
  end.
```

RUN

```
This is loop number 1.
This is loop number 2.
This is loop number 3.
This is loop number 4.
This is loop number 5.
This is loop number 5.
This is loop number 4.
This is loop number 3.
This is loop number 2.
This is loop number 1.
```

ANALYSE

For loops are demonstrated perfectly in this example program. Now, take out your tools, it's dissection time!

- ***Line 4** - This declares the variable i*
- ***Line 7** - This loops i from 1 up to 5*
- ***Line 8** - This line prints out a message with the value of i*
- ***Line 9** - This loops i down from 5 to 1*
- ***Line 10** - This prints exactly the same thing as Line 8.*

As you can see clearly above, the loop is repeated from 1 up to 5 and 5 back to 1 again. Well, that's nice, but what if you want a sort of mix between the two? You know, repeat something until a condition is satisfied, or while a condition is true... Well, what you're looking for is just up next! 'Repeat-until' and 'while' loops do just that, so read on!

3. Repeat-Until and While Loops

Repeat-Until loops repeat a sequence of code until a condition is satisfied. For example, you can say: *repeat* tearing up the documents *until* there are no more documents left. That can be directly translated to Pascal code.

```
repeat
  x := x - 1;
until x = 0;
```

This code will loop again and again until x is equal to 0. In other words, x would be decreased by 1 repetitively until it is 0. Yes, I can see that look on your face. You're thinking of ways to make use of this, aren't you? Clever boy!

Rem - In case you're desperate for accuracy, you'd like to know that Pascal actually states that you are to leave out the semicolon on the last line of any code block. That also includes the repeat-until loop, because that code between 'repeat' and 'until' is also considered a code block! Placing a semicolon however, still works, because Pascal 'adds' an extra empty line without a semicolon at the end. And yes, it is recommended for you to place semicolons anyway. Better than griping around later adding strain onto your keyboard!

Now, about the while loop. The while loop is a very close cousin of the repeat-until loop and works in almost the same way. The loop will repeat itself only *while* the condition is true. Once the condition becomes false, the loop exits. Here's how it looks like:

```
while x <> 0 do
  x := x - 1;
```

This is a variation of the repeat-until loop above, except that it checks if x is **not** 0. If that is true, it will decrease x by 1. However, once x reaches 0, the loop exits and continues with the rest of the code.

Here, some of you may ask, what's the difference between the two loops? Well, for the repeat-until loop, *the code will be run at least once*. In other words, no matter what the condition is, the code will be run first, then the condition tested. For the while loop however, the test is carried out first before deciding whether to execute the code. Thus, the code may not be run at all.

Well, below is an example program to demonstrate these two loops. Pay close attention to what happens, and try to figure out why.

TYPE

```
1: program LoopADoop2;
2:
3:   var
4:     x: integer;
5:
6:   begin
7:     x := 1;
8:     while x <= 5 do
9:       x := x + 1;
10:    Writeln(x);
11:
12:    x := 1;
13:    repeat
14:      x := x + 1;
15:    until x >= 5;
16:    Writeln(x);
17:  end.
```

RUN

```
6
5
```

ANALYSE

Now, let's analyse this. What do you think makes the answers different? Well, let's figure out why. (Better than leaving you alone in the dark...)

- ***Line 4*** *- Declares variable x*
- ***Line*** *7 - Assigns 1 to x*
- ***Line 8, 9*** *- Ah, here's our first loop. First, the loop tests if x is* ***less than or equal*** *to 5. If so the loop runs the code. So, if you trace the loop, you'll see that the loop repeats five times, with the final value of x being 6. Ta-da!*
- ***Line 10*** *- This prints out the value of x*
- ***Line 12*** *- Resets x*
- ***Line 13, 14, 15*** *- Now, you may have noticed that the difference between this loop and the while loop is that while the while loop (pun intended!) provides a condition to* ***continue*** *the loop, the repeat loop gives a condition to* ***end*** *the loop. Thus, if you trace the loop again, once x reaches 5, the loop stops instead of continuing* ***one more time*** *like the while loop! Amazing!*

So, now you know these two very simple loops, it's finally time to mention our last and final control flow loop - the case statement. Pretty complicated, but very useful in future!

4. The case statement

The case statement is very similar to multiple if statements. In fact, it was created to replace messy code caused by to many *ifs*! The case statement takes a value, looks it up against a list, and if it finds an equivalent value, it will run the code associated with it. If the value is not found, it runs the default code, if there is any. Confusing? Em, it'll be better if you get a look at how the case statement looks first:

```
  case x of
    1: Writeln('X is equal to 1');
    2: begin
         Writeln('Did you know?');
         Writeln('X is equal to 2');
       end;
  else
    Writeln('X is not equal to 1 or 2!');
  end;
```

As you can see, the case statement looks at x, checks it against 1, then 2, and if x is not equal to any of those, it prints "X is not equal to 1 or 2!". Fantastic! Note that you can leave out the else part in a case statement. Without it, when no match is found, the case statement will do nothing. Ahh, easy does it.

Some of you may be wondering though, what if there are two values x can be equal to? Which one will it run? For example:

```
  case x of
    1: Writeln('X is equal to 1');
    1: Writeln('1 is equal to x');
  end;
```

Well, the case statement will print the first line but not the second. **The case statement only executes code attached to the first matching value.** The rest are history. However, that's not the only thing you need to remember about the case statement. Here's are some *case* pointers:

1. The values you input for case to match **must** be constants. You cannot ask, for example, for case to match x with another variable like y. However, it can work with a numeric constant you had declared or an explicit numeric number.

2. You can only use case with numbers or characters, nothing else. That means no strings are allowed! (Wait a minute, I haven't even used a string yet!)

3. You can attach code to more than one value in a case statement. For example, if you want a certain piece of code to execute when the variable matches 1 to 50, you can type '1..50'. That's a 1, two full stops, and a 50. You can also specify different values at one go by seperating them with commas. Example: '1, 5, 8' will match a variable if it is either 1, 5, or 8.

4. No, case has nothing to do with Long John Silver's treasure chest.

I bet you're swirling with questions right now, so here's your final example program! Good luck! Here we go...

TYPE

```
program TheMysteryCase;

  var
    x: integer;

  begin
    x := 5;
    case x of
      1..50: Writeln('X is between 1 to 50');
      51..100: Writeln('X is between 51 and 100');
    else
      Writeln('X is off the range of 1 to 100!');
    end;
  end.
```

RUN

```
X is between 1 and 50
```

ANALYSE

This is fantastic! Here the case statement is used to check whether x is between 1 and 50 or 51 and 100. This is how it works:

- ***Line 4** - Declares variable x*
- ***Line 7** - Assigns 5 to x*
- ***Line 8** - This line starts the case statement to test x*
- ***Line 9** - If x is between 1 and 50, it will print the appropiate message*
- ***Line 10** - If x is between 51 and 100, it will print the appropiate message*
- ***Line 12** - If x does not fit both coditions, it will print this message*
- ***Line 13** - This ends the case statement*

Simple, easy, sweet. That's how the case statement works. You might have noticed that this program could have been written simply using one if statement, but hey, this is just an example!

The End Of It All

This marks the end of Day 4. Next up, is Day 5, Procedures and Functions. Ahh, that was a long one. Loosen up, you're going to learn something new!

Class B
Day 5 - Procedures and Functions

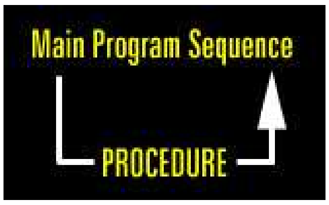
What you are going to learn today:

- What procedures and functions are
- What's scope of variables
- How to implement procedures and functions
- And an example program to round it all up

Procedures and functions are the topics today, so listen up! After going through this tutorial, you will never want to give up programming again!

What are Procedures and Functions?

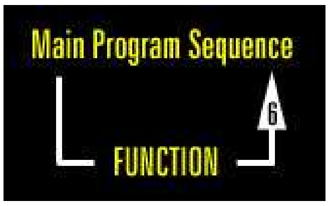
Procedures? Functions? What? Is this some sort of calculator manual? No, procedures and functions are a feature of Pascal which allow you to repeat a certain piece of code or calculation again and again. Isn't that great? But first, we must find out what procedures and functions are in reality.



First of all, what is a procedure? Well, let's say you've got a job as a dishwasher. The process of washing a dish could be:

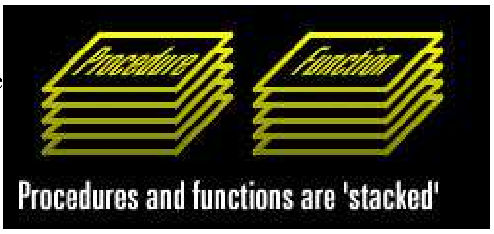
1. Dip the dish into the water
2. Cover every inch of the dish with soap
3. Rinse and dry the dish

So, every time you need to wash a dish, you do just that. Dip, soap, dry. Dip, soap, dry. Dip, soap, dry. Even when you go home, you dip, soap, dry. Dip, soap, dry. It's the same sequence! A procedure works the same way. When people give you instructions, they tell you to "wash the dishes", not "dip, soap, then dry". This is because you already know what is involved when you wash dishes. A procedure is just a faster, and more modular way, to get the job done. By replacing a stack of instructions with one single statement, if makes code easier to read and debug.



Now about functions. A function is just like a procedure except that it returns a value. For example, somebody may ask you to count the number of chairs in a hall. You first step in, count, then return the number of chairs in the hall back to the person who first asked you. That is a function. A function simply returns another value back into the program, such as complex calculation results, the position of the mouse cursor, or the number of times an atomic bomb had dropped onto the Earth. Functions can report almost anything. Numbers, strings, characters, anything! And if you want to, you can use functions replace procedures completely! Isn't that great?

Scope of Variables



Now that you understand the concepts, let's be honest. Everything good in life has a catch. Procedures and functions are no different, and this catch is the concept of *scope*. To understand this concept, visualise a stack of exercise books on your table. Everytime you call a procedure or a function, it will be placed at the top of the stack. When the procedure or function finishes, it is removed. Now, when a procedure and function goes, everything else declared in it goes too! That means that if you declare a variable in procedure x, you can't use that variable in procedure y! The variable simply goes poof, and disappears into a mist of sparkling electronic dust. However, you can use variables which were declared in lower procedures and functions in the stack. Unbelievable! Confusing, maybe, but *very* powerful...

How procedures and functions are implemented

How are procedures and functions implemented? Well, here's the low-down on the syntax for declaring your very own procedures and functions.

Procedures are declared using the procedure keyword, as you can see below. The name of the procedure comes first, followed by a semicolon, then the contents of the procedure.

```
procedure WhyMe;

  var
    i: integer;

  begin
    for i := 1 to 10
      Writeln('Why Me?!');
  end;
```

A procedure is just like a separate program in itself. You can declare variables and use them within your program code. However, as described above, after the procedure ends, the variables vapourize with it. So, now that you have declared a procedure. How do you call it? Ahh, this is easy. Just type in the name of the procedure. What else could be easier?

```
begin
  WhyMe;
  Writeln('That was my true feelings regarding Exams');
end.
```

When this program executes, it will print 10 "Why Me?!"s and print the final message "That was my true feelings regarding Exams". After a procedure finishes, it returns back to place where it was called. Isn't that smart? Pretty innovative.

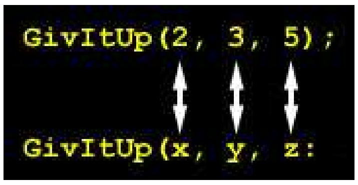
Functions, on the other hand, cannot be used just like that because they return a value back to the program instead of just doing things. For instance, we may have a function which acts like this:

```
function AddEmUp : integer;
  begin
    AddEmUp := a + b + c;
  end;
```

This function will return the sum of a, b, and c. Well, how does it work? The function name comes after the keyword 'function'. After the function name, you get the data type the function should return. Huh? Yes, even when using functions, you can't run away from the nearly limitless list of data types. The value the function should return is assigned to the name of the function. In this case, because the name of the function is 'AddEmUp', we assign the result of the function to the same name! How do you use functions then? Well...

```
begin
  a := 5;
  b := 9;
  c := 4;
  Writeln(AddEmUp);
end.
```

This program will print out the sum of a, b, and c! Marvellous use of functions, isn't it? Well, it isn't finished yet!


```
GivItUp(2, 3, 5);

GivItUp(x, y, z:
```

Procedures and functions would have been dreadfully boring and unflexible if it wasn't for a great feature called *parameters*. You can use parameters to pass values to the procedure or function to use. For example:

```
function AddEmUp2( x, y, z: integer ) : integer;
  begin
    AddEmUp2 := x + y + z;
    end;
```

Then, to use this function, you must provide values for x, y, and z. This is done by using brackets.

```
  begin
    Writeln(AddEmUp2(2, 3, 5));
  end.
```

In this example, x would be 2, y would be 3, and z would be 5. You're passing values over for the procedure or function to make use of. Got the idea? No? Well, here's another example.

```
  procedure PrintEm( x, y, z: integer );
    begin
      Writeln(x);
      Writeln(y);
      Writeln(z);
    end;
```

If you pass the values 2, 3, and 5, this procedure would print out those numbers. That's great! Now you can provide information to the program on the fly. Power on the go...

The final example program...

Here it is, in its full glory, the example program of the day! Buckle up, and enjoy typing this utterly simple yet powerful program.

TYPE

```
program AddEmUpAgain;

  function AddEmUp( a, b, c: integer ) : integer;
    begin
      AddEmUp := a + b + c;
    end;

  procedure PrintIt( a, b, c: integer );
    begin
      Writeln('The sum of a, b, and c is ', AddEmUp(a, b, c), '.');
    end;

  begin
    PrintIt(2, 3, 4);
  end.
```

RUN

```
The sum of a, b, and c is 9.
```

ANALYSE

Although it looks complicated, this is actually a very simple program. It does absolutely nothing but to add three numbers together! Is that dumb or what?

- ***Line 3*** *- This declares the function AddEmUp, which takes three parameters and adds them up.*
- ***Line 5*** *- The only line of AddEmUp, it justs returns the sum of parameters a, b, and c.*
- ***Line 8*** *- Declares procedure PrintIt, which also accepts three parameters.*
- ***Line 10*** *- This line simply prints out the correct message by calling AddEmUp and passing it the three parameters.*
- ***Line 14*** *- This single line starts the whole ball rolling!*

Well, that was the great example program of the day! But this topic is far from over. Here are the tips and rules we couldn't fit into the text without making it ridiculously boring. (As if it wasn't already...)

1. You must declare a function or a procedure before it is used. For example, in the example program above, you can't just put the procedure on top of the function because PrintIt needs to use AddEmUp, it needs to be declared first.
2. However, if the world somehow gives you a reason to use a procedure before you declare it, you can use the **forward** keyword. Find out from your nearest F1 key!
3. What happens if you provide more parameters than a procedure or function needs? Well, when you do so (or vice versa), the compiler simply gives you an error, so go on and experiment! (Em, don't play too much though...)
4. No, you can't re-declare constants in a new procedure - Why would you anyway?
5. Procedures and functions are a very important part of *structured programming*. Suffice it to say that is you don't learn structured programming now, you'll never make it big in the future.
6. No, there will not be another Bill Gates analogy for this tutorial.

There's a lot more - but hey, this is for beginners right? So, that's it, the list of important things for procedures and functions! Ahh, now you know why people invented lists...

The Final Chapter is coming

Joy! Rapture! Divinity! (Get the joke?) Day 5 is finally over and you can go on to Day 6 - Comments and Other Neat Stuff. Neat Stuff? What neat stuff?! Well, find out for yourself. Avé!

Class B
Day 6 - Comments and Other Neat Stuff

What you are going to learn today:

- What comments are and how to use them
- How to use units (not make them)
- How to format Writeln text
- How to get information from the user

Relax. This is the most mixed tutorial in Class B - that means that you don't need to concentrate on anything! I mean, what could be more fun than a mess of topics to learn about?

Comments? What comments?

Comments are so darn simple, we wonder why people talk about them at all. Well, simply put, comments are just code the compiler never executes. This lets you type *text* into your code - because the compiler never executes it. Comments are used to annotate code, or just to take notes. For example:

```
  a := w * h;  { Width times Height }
```

The words "Width times Height" don't even get a glance from the compiler! To mark out a comment, you use the two curly brackets - { and }. Comments can be almost anywhere, but for goodness sakes, don't put them inside the middle of a statement. Comments can also be used to temporarily mark out statements.

```
  { This code is worthless.
  a := a;
  See what I mean? }
```

The statement a := a is never executed at all. Aha! Now you see the power involved. Imagine, with just two characters, you can render an entire sequence of code useless. Is that power or what?

How do you use units?

Pascal has a few built in commands - Writeln, and so on. But to do more complex things, like changing the colour of the text or repositioning the cursor, you need different commands which aren't built-in. This is exactly what units are for. Units are files which actually define *new procedures and functions* for you to use! Amazing! Pascal generally comes with a few standard units which are listed blow:

• Crt	• Dos	• Graph
• Graph3	• Overlay	• Printer
• System	• Turbo3	• WinDos

Of all those units above, the one you'll probably use most would be Crt. It provides procedures to change the text and background colour and to reposition the cursor - Yes, sounds very well and good, but how do you include a unit in your program? Well, you've learnt a few keywords: *const* and *var*, so here's another one: **uses**! To include a unit in your program, type this before your code block, just like how you would declare variables or constants:

```
  uses
    Crt;
```

This way, the Crt unit is loaded and all its procedures and functions will be available to you! If you want to load many units, separate them with commas. Just make sure there's no typo, and off you go!

- You've probably realised something by now, in any Pascal program, besides the program code, there are a few 'sections' before the block. The const, var, and uses sections (that's not all though, get ready for *type*...) declare variables, constants, or load units. You may be wondering, what order should you put them? Well, it really doesn't matter, but just remember, before you use anything, you must first declare it. Thus, if your var section needs something from the const section, the const section must of course come before var. Here's my suggestion: first should be *uses*, then *const*, and finally *var*.

Now, you can go on to use the TextColor and GotoXY procedures defined in Crt - just like any other function. (Note that these procedures will not work if Crt is not loaded.) So you can type this:

```
  TextColor(Blue);
  GotoXY(10, 10);
  Writeln('SKCUSTISOCBVETAHI');
```

Those lines would print out "SKCUSTISOCBVETAHI" in blue at screen location 10, 10. (Crack the code! Find out what SKCUSTISOCBVETAHI means and win a $10 vouncher!) Isn't that great! Complete control! For a complete list of Crt Procedures and Functions, type Crt and press Ctrl-F1. Select "Crt-Procedures" to view a list of all the procedures and functions in Crt. Nice work.

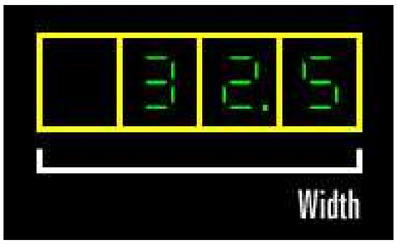
How do I format Writeln?

For many of you who decided to jump the gun and play with non-integer data types (numbers which can have decimal points), you may have noticed that your output was far from what you expected. For example:

```
  var
    RealNum: real;

  begin
    RealNum := 7.3;
    Writeln(RealNum);
  end.
```

Upon running this program, you find out that your expected 7.3 never appeared the way you wanted it. It became "7.3000000000E+00" - pure trash except for true mathematicians like us. (Ahem) Well, what's the problem? Ahh, it's all very simple. When you print out a *real* number, its *precision* is well beyond the limits of normal human understanding. What it actually tried to print out was "7.30000000000000000..." until we all fell flat staring at zeros. To solve this problem, you must understand the concepts of *width* and *digits*...

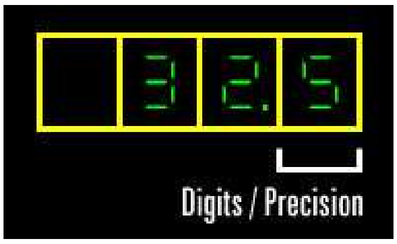
Width is the mininum width the number must take up, and this value is set using a colon next to the value name. Assuming x is 6784, the following statements give the following results:



```
  Writeln(x:3);     6784
  Writeln(x:4);     6784
  Writeln(x:5);      6784
  Writeln(x:6);       6784
```

As you can see, whatever value x is, you know that the number will always take up that number of spaces. However, if the width is lower than the width of the number itself, the width is expanded automatically, as shown in the very first example: Because 6784 has 4 digits but we gave a width of 3, the 6784 will still be printed out in full. However, if the width is larger than the length of the number, the number will be aligned to the right, and spaces will be added to the left.

However, that's not enough. The result after using Width still gives a number with an 'e' in the middle. Ahhh, what now? Precision! That's it! We tell Writeln how many digits we want after the decimal point! And how do we do this? By adding another colon after the Width, followed by the precision:



```
  Writeln(x:5:3);     3.142
  Writeln(x:8:3);        3.142
  Writeln(x:8:5);      3.14159
```

Notice that the number is not truncated, but rounded off. Ahh, intelligence of Pascal, I believe? (Now you know why I love this language!) With this information, Numbers can be printed out in the exact way you want them to be!

How to get info from the user

You might have noticed that we've been making programs which are kinda one-sided. The user has no active participation in the whole thing. Thus, the user feels left out. So, in order to keep your users smiling, use the Readln command, as illustrated below:

```
  var
    x: integer;

  begin
    Writeln('Enter Value...');
    Readln(x);
  end;
```

Simply put, this program just stores what you typed into x - a very interesting move. Try it out. It works, right? Well, that's not all. Readln can read more than one value at a time:

```
  Readln(x, y, z);
```

Used this way, you can either separate each value with an enter or a space. Oh, how wonderful this is! (More of this I'll go singing like The Sound of Music) You've finally finished Class B!

Our Final Word

Now that was hectic, wasn't it? Don't worry, you've only got two more classes to go. After all, *this was the longest class*. So go, get your newest video game or something, play it for 36 hours non-stop (I personally play SimCity 2000, but naa, you wouldn't), take a 48-hour sleep, and come back for Class I, ready and going!

Class I

Here we go again, splashing further, but for a shorter period. Only three tutorials this time, but these are going to be enough to make you dizzy (and to give us CTS for a month), because of their subtle complexities. Once again, you can download the code samples here. Good Luck, and enjoy!

***Day 7* - Strings and Characters**

Strings and characters, two things we all understand very well simply because we're making use of them everyday. In this tutorial, we talk about how to use strings, and why Pascal strings are a whole lot better than those menacing C strings!

***Day 8* - User Defined Types**

Probably the most confusing tutorial in Class I, this teaches you something only 12.7% of the beginners know (but don't understand). As if the data types we have are not enough, Pascal lets you make your own - Oh No! Look Out! Arrays, Records, and Sets are coming your way!

***Day 9* - Pointers and Such**

Memory is the buzzword here, and if you have a headache, don't come near. This tutorial teaches you the basics, and what you can do with pointers. Be careful, this is the *second* most confusing one. Linked Lists, Binary Trees, how could this be worse?

Class I

Day 7 - Strings and Characters

What you are going to learn today:

- What strings and characters are
- How strings are implemented in Pascal
- And the final example program!

Welcome back! This time, we're going to learn about strings and characters - storage space for text. Don't worry, this has nothing to do with *real* strings found on your sewing machine (put your needles away!), so just sit back, relax, and enjoy this tutorial.

Strings and Characters

Unless you're a caveman from the Stone Age and has never seen a line of text, you should be able to identify with strings and characters easily. You all know the integer and real types, so here's another two - enjoy!

"This is a string."

Strings are text. Simple as that. I mean, just as how integers can store the value 15, strings can store things like "Life is like a box of chocolates", or "What, me worry?" Strings can contain any line of text 255 characters long or less. Any longer than that and the string will be truncated, so be forewarned! You can also use strings to store names, keywords, or just plain text. And as you'll soon find out, strings can be highly flexible things, so be careful!

"A", "B", "C"

On the other hand, characters don't store text. They can only store *single characters*, such as "a", "b", or "C". (Note that uppercase characters are different from lowercase ones.) This is highly useful, if you don't want to make one big string just to store one character. Imagine, you're going to make a box to store 255 teddy bears, but you're only going to put just one. Now, is that stupid or what?

How does Pascal implement them?

Ahh, C programmers would be happy to note that there are no ridiculous commands to memorise just to initialize a string. This is because the Pascal string design is much simpler. Instead of having some crappy null-terminated string, (don't worry, no technical details) Pascal has a string which contains the length of the string in the first byte. Yes, it may sound a little technical, but don't worry, you don't have to know *that* to use Pascal strings.

To start off, we'll first tell you how to use a string. In case you start thinking that you need to type mounds of code just to use a string, well no, all you have to do is to just declare a variable with the data type 'string', like this:

```
var
  AString: string;
```

Uh-huh, that simple. And yes, how do you use them in code? Well, Pascal uses the apostrophe to mark out strings, just like how you typed text for the Writeln command. So, to assign AString the value of 'Sim Products Forever!', you type:

```
begin
  AString := 'Sim Products Forever!';
end.
```

Yes, now AString contains that wonderful line of text. Isn't that great? You can retrieve the value any time you want now - and yes, you can even add strings together to form one big string, like this:

```
  begin
    AString := 'Sim Products Forever!';
    AString := AString + ' Isn''t that great?';
    Writeln(AString);
  end.
```

Rem - Two things to note: Firstly, notice the the line ' Isn't that great?' turned into ' Isn''t that great?' Why did the single apostrophe turn into a double one? Well, because the apostrophe is also the marker to tell Pascal that you want to end the string, to include an apostraphe *itself*, you would need to place double ones. Second, what if you want to create a string which can store less than 255 characters? (which is rarely the case unless you're really out of memory) Well, you just simply add a number representing the length when you declare the string - **AString: String(50);** That line will declare a string which can only store up to 50 characters, after which, everything extra will be truncated!

Now for characters. The case is mostly the same - just that you can only store one character only now. The data type for charcters is 'char'. Thus, (with an air of gusto), a typical character declaration looks like this:

```
  var
    AChar: char;
```

Yes! Magic is in the air! Now, we can give AChar a value the same way we do it with strings. Don't give a char a string value though, you'll get the infamous 'Error 26: Type mismatch.' error message from the compiler.

```
  begin
    AChar := '*';
  end.
```

Yep. That's it. The end of the story. We have a character holding the value '*', and voila! You have a perfect example of using a character. Not only that, but our simple introduction is over too! Well almost. Just below is the example program of the day. Happy Typing! (Hint of sarcasm)

The Example Program (Sacred Words)

As what the great poet Kasbury Hermes once said, "Strings are used to tie the characters of our lives together.", we are going to have an example program to do just that. (For you literature students flipping through your reference books there, you'll like to know that there's no such poet and no such quote. They were all products of my imagination, obviously.) This example program is so simple, it's going to bore you to death...

TYPE

```
program StringChar;

  var
    Stringy: String;
    Charey: Char;

  begin
    Stringy := 'Animaniacs';
    Writeln(Stringy);
    Charey := 'Z';
    Writeln(Charey);
  end.
```

RUN

```
Animaniacs
Z
```

ANALYSE

What's there to say? What's there to explain? What's there... oh never mind. Here's the explanation for your personal reading pleasure:

- ***Line 4** - This line declares the string Stringy*
- ***Line 5** - This line declares the character Charey*
- ***Line 8** - This line assigns Stringy the string "Animaniacs"*
- ***Line 9** - This line prints Stringy*
- ***Line 10** - This line assigns Charey the character 'Z'*
- ***Line 11** - And finally, this line prints out Charey!*

Ending Words

Yes! You've finished Day 7! Now, that was simple, wasn't it? Well, get ready, 'cos it's gonna get complicated. Lift up your head, breathe hard, and go straight into Day 8 - User Defined Types.

Class I

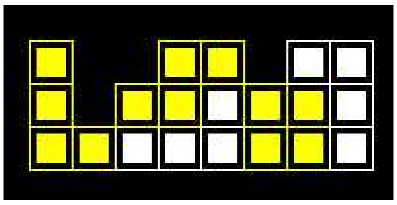
Day 8 - User Defined Types

What you are going to learn today:

- What user-defined types are
- How do we implement user-defined types
- The different types of user types
- And what this has to do with you

Horrors of horrors, you are going to learn how to make your own data types! Well, not really. Your so-called 'data types' are actually just your own customised way of storing values, as you'll soon find out. Feeling curious? Well, read on!

What are user-defined types?



Simple question. Imagine you have these wonderful data types, but you're getting messed up figuring out which variable belongs to square 2,3 and which belongs to square 7,1. (That is assuming you live by creating brick games, of course) Bricks start falling backwards and invert instead of rotating - with you going crazy inside! But you can't do anything, simply because you've got so many variables that you can't figure out which one is which. (Was that the variable representing its current X position or what?) What you really need is just a *better way to store your values*, and in this case, you could replace the entire game board with one *array* variable, saving you both time and sanity. (In fact, I have confirmed suspicions that Tetris is just one big array manipulating program...) What? You don't know what's an array? Well, don't worry, you'll find out soon. Hey, Polish writer Junis Bright once wrote: "Patience ends all waiting" (another fake quote), so just hang on, okay?

How do we implement User-Defined Types?

You may not know any of the user-defined types yet, but before we can discuss them, we have to know how to use them, right? (You don't give a person a PC without the manual right?) There are two ways to declare User-Defined Types:

First, you can declare it in a normal var declaration, with the data type representing what you want it to store. We'll use a known example - the string type:

```
var
  ASmallString: String(10);
```

This declares a string with 10 characters, but wait! There are two problems with this method: First, it is highly irritating to type 'String(10)' everytime you want to declare a variable with the same type. Second, 'String(10)' has no meaning on its own - wouldn't ir be better to call it 'TenCharacters' or something? Well, here's where the second method comes in!

You've learnt *uses*, *const*, and *var* right? Well, here's *type*! The type keyword can replace a data type with another name:

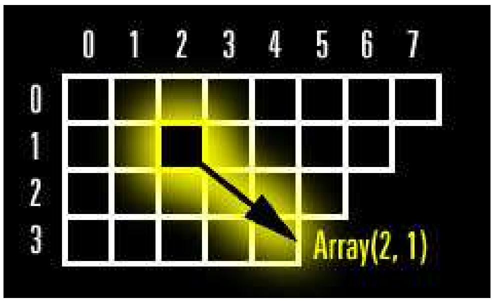
```
type
  TenCharacters = String(10);

var
  ASmallString: TenCharacters;
```

As you can see, type works almost the same way as const. TenCharacters is now the same as String(10)! Yes, that is true power. You can replace any data type with any other legal variable name. Ooo, ahh, that's how it works...

The different User-Types follow...

Note that these aren't all the user data types available, (we originally had all of them here, but halfway through we realised we were *well behind schedule*...) These types are only those which we feel are absolutely neccesary, such as records, sets, and arrays. Besides that, no way!

1. Arrays



Can you imagine making a Game of Life program and having to declare a variable for each and every cell? Not only would it be terribly confusing, it would require a Computer Science degree to make it efficient enough to use. So, what do we do? We use arrays! Arrays are like dimensional tables - A one-dimensional array is like a sequence of values, a two-dimensional array is like a table of values, and a three-dimensional array is like a cube of values. A four-dimensional one is like a, em, never mind. Arrays are much simpler than what they sound, it's just that it's so abstract that this is the only logical way to explain it - don't believe me? Well, let's see an example of an array declaration to prove it:

```
type
  GameGrid = array [1..32, 1..32] of integer;

var
  Grid1: GameGrid;
  Grid2: array [1..32, 1..32] of integer;
```

In case you're wondering, Grid1 and Grid2 are exactly the same. Now how does this array work? Well, you can say that the first number is the X coordinate and the second number is the Y coordinate. Thus, cell 3, 1 is represented by Grid1[3, 1]. Note that those are square brackets, not curly or round ones. And yes, you must also specify the data type for the entire array when you declare it. Want more examples? Ahh, here you go:

```
var
  Line: array[0..255] of integer;
  Table: array[0..255, 67..127] of char;
  Cube: array['A'..'Z', 1..10, 1..10] of real;
```

Now you see why the dimensional method is always the most logical way to explain arrays. Also, note that you can also use characters to specify a dimension. In such a case, you refer to the array like this:

```
begin
  Cube['S', 5, 5] := 70;
end.
```

In the statement above, you assign cell 'S', 5, 5 the value 70. Fantastic, isn't it? Now, get ready for another example program! This ones simple, but is fantastic at clarifying any doubts you have in your mind.

TYPE

```
program ArrayCrazy;

  type
    TTable = array [1..2, 1..5] of integer;

  var
    Table: TTable;
    a, b: integer;

  begin
    for a := 1 to 2 do
      for b := 1 to 5 do
        begin
          Writeln('Input value for ',a ,', ', b,' : ');
          Readln(Table[a, b]);
```

```
16:         end;
17:     for a := 1 to 2 do
18:       for b := 1 to 5 do
19:         Writeln('Value ', a, ', ', b, ' is ', Table[a, b]);
20:   end.
```

RUN

```
Input value for 1, 1 :
1
Input value for 1, 2 :
2
Input value for 1, 3 :
3
Input value for 1, 4 :
4
Input value for 1, 5 :
5
Input value for 2, 1 :
6
Input value for 2, 2 :
7
Input value for 2, 3 :
8
Input value for 2, 4 :
9
Input value for 2, 5 :
10
Value 1, 1 is 1
Value 1, 2 is 2
Value 1, 3 is 3
Value 1, 4 is 4
Value 1, 5 is 5
Value 2, 1 is 6
Value 2, 2 is 7
Value 2, 3 is 8
Value 2, 4 is 9
Value 2, 5 is 10
```

ANALYSE

This program may look a little complex, but actually it's very simple. If you're having problems visualising, we suggest that you change the default colours under Options, Environment, Colors... to make the different elements of code clearer. Let's begin!

- ***Line 4*** *- This defines the type TTable*
- ***Line 7*** *- Declaration of variable Table of type TTable*
- ***Line 8*** *- Declares two counters, a and b*
- ***Line 11*** *- This starts a loop for the X coordinate*

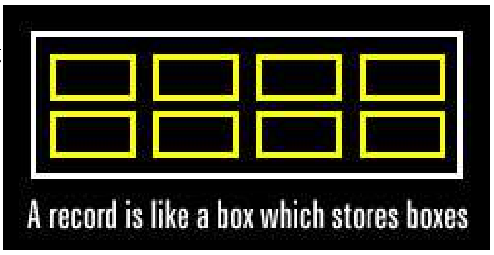
- ***Line 12*** *- This starts a loop for the Y coordinate: notice that this is a for loop in a for loop, this loop must finish before the first loop can loop again. Programmers call these loops* ***nested loops****.*

- ***Line 14 & 15*** *- Prints a message and inputs what you type into the array at the specified coordinates.*

- ***Line 17 & 18*** *- Starts another nested loop.*

- ***Line 19*** *- This time, the loop prints out the value of Table at the specified coordinates. Ta-da! The End!*

2. Records



Imagine you're the principal of your school. Now, you're supposed to keep track of the Name, Class, Height, and Weight, of all those 15,000 little brats running around driving your teachers crazy. Well, how are you going to do it? Ahh, isn't this simple. Let's just create a big, gigantic array to arrange all this information. Or even better, let's make 4 separate arrays to store each detail about each brat in your school! Of course, the above are very possible, and very feasible feats, but the more intelligent, and logical, choice would be to use a record to represent each brat. Well, what's a record? In a way, a record is like one big variable holding lots of other variables. Something like one big box holding lots of boxes. Thus, each brat could be stored as a record with variables representing his name, class, height, and weight - a perfect conclusion to an imperfect education. Now, how does a record look like? Well, here it is:

```
type
  Student = record
    Name, Class: string;
    Height, Weight: integer;
  end;
```

In this example, we have a type Student which contains the variables Name, Class, Height and Weight inside. The so called 'Little Boxes' are encased inside the 'Bigger Box' between the keywords *record* and *end*. How to declare a record variable then? Look:

```
var
  Brat: Student;
```

It's essentially the same method you would use to declare normal variables, isn't it? Well, here's question No. 2! How would you refer to the little boxes inside the big giant box? Ahh, you simply just use the *dot operator*! For example, you want to refer to the 'Little Boxes' inside variable Brat:

```
begin
  Brat.Name := 'Lee Yang Yang';
  Brat.Class := '1M';
  Brat.Height := 12;
  Brat.Weight := 6;
end.
```

What you type would be the name of the variable, followed by a dot, then the name of the 'Little Box' you want to refer to. Yes, you have successfully contained all the bio-data of one student into one little variable! Fantastic! Do you see how it works now? Brat is the name of the big box, Name, Class, Height, and Weight are the names of the littles boxes. Great, eh? But that's not all - Many of you may find using the record variable name again and again irritating, (especially after you're suffering from CTS). See?

```
begin
  { How many times do we type "Brat"? }
  Brat.Name := 'Lee Yang Yang';
  Brat.Class := '1M';
  Brat.Height := 12;
  Brat.Weight := 6;
end.
```

Now isn't that long-winded. Hmm, one shortcut you can use to shorten this is the with keyword, which lets you refer to any of the 'Little Boxes' without typing the variable name. Ahh, isn't that nice? No more extra strain on your keyboard. Here's how it looks - the with statement states the name of the record first before you use its members.

```
with Brat do
  begin
    Name := 'Lee Yang Yang';
    Class := '1M';
```

```
      Height := 12;
      Weight := 6;
    end;
```

See, the word 'Brat' only appears once! No extra 'Brats' to block your way - True efficiency! And yes, that also marks the end of our introduction to records. There's only one thing left to do: an example program. Well, you can't run away from it, so why not type it?

TYPE

```
1: program RecordingTime;
2:
3:   type
4:     Student = record
5:       Name, Class: string;
6:       Height, Weight: integer;
7:     end;
8:
9:   var
10:    TheBrat: Student;
11:
12:  begin
13:    { We first assign TheBrat values }
14:    TheBrat.Name := 'Augustus Gloop';
15:    TheBrat.Class := '5F';
16:    TheBrat.Height := 145;
17:    TheBrat.Weight := 67;
18:
19:    { Now we print them out! }
20:    with TheBrat do
21:      begin
22:        Writeln('Name: ', Name);
23:        Writeln('Class: ', Class);
24:        Writeln('Height: ', Height);
25:        Writeln('Weight: ', Weight);
26:      end;
27:  end.
```

RUN

```
Name: Augustus Gloop
Class: 5F
Height: 145
Weight: 67
```

ANALYSE

This is a simple one. (Then again, I say that for all the programs, don't I?) What it does is to simply input some values into a record and print them out again. Here's the final breakdown:

- ***Line 4, 5, 6*** *- Defines type Student with Name, Class, Height, and Weight. (Bet you've seen this before!)*
- ***Line 10*** *- Declares variable TheBrat as type Student.*

- ***Line 14, 15, 16, 17*** *- Here, we gave TheBrat some values, the hard way. See how irritating repeated typing can be? Oooo... (That tingling sensation!)*
- ***Line 20*** *- This with statement refers to record TheBrat*
- ***Line 22, 23, 24, 25*** *- Finally, these lines print out the values of TheBrat the easy way. Ahh, see that we can leave out 'TheBrat' now? So relaxing! We're finished for now, so let's go on to type No. 3*

3. Set Types

If you ever need a virtual check-list, sets are the closest you'll ever get. What sets do is to provide a list of objects, no more than 255, for you to 'check' on and off. As you continue learning Pascal, you'll find that sets can be pretty useful instead of a whole long list of Boolean variables - for obvious reasons.

- Opps! Of all the data types, we still haven't introduced Boolean yet, haven't we? Well never mind, we'll explain it now. You see, a Boolean type is just a type which is either *true* or *false*. Thus, you can type things like 'aBoolean := a = b;' - if a is equal to b, aBoolean is true. Otherwise, it's not. That simple!

Excited yet? Well, you may be wondering how do you declare sets then? Patience, patience! These things are not meant to be rushed. Ahh, the set definition looks like this:

```
type
  aSet = set of 1..255;
  aSet2 = set of Char;
  aSet3 = set of ( Alpha, Beta, Gamma );
```

In this example, aSet has 'Check-Boxes' labelled from 1 to 255, and aSet2 has 'Check-Boxes' labelled with the different characters from the ASCII table. Notice aSet3 - Alpha, Beta, and Gamma are not any type of value, but just names for the 'Check-Boxes' in the set.

- Wait a minute! Remember I said that a set can only 255 values? How come I can declare a set with type Char - (aSet2 = set of Char;)? Well, simple. If you look up the ASCII chart, you'll notice that there are *exactly 255 characters* available (minus the null chracter, of course.) What the set does is to use all those characters as 'Check-Boxes', see? Don't try this with integer or real though, there are just too many possible values.

Now that you know how to declare sets, how to you tell it what to check on and what to check off? Well, there are two very simple ways, depending on what you want to do:

The first way is very simple - you place the values you want to turn on inside square brackets and assign it to the set, like this:

```
begin
  aSet := [1, 7, 10..12, 5];
  aSet2 := ['A', 'K', 'X'..'Z'];
  aSet3 := [Alpha];
end.
```

For aSet, values 1, 5, 7, 10, 11, and 12 will be turned on. For aSet2, values 'A', 'K', 'X', 'Y', and 'Z' are turned on instead. aSet3 just turns on 'Check-Box' Alpha. See how simple this whole thing works? Innovative, isn't it?

The second method to assign values is to use *set operators*: +, -, *. Unlike the plus, minus, and multiply functions, these are the add, minus, and intersect values! What? Well, here are the three set operator rules:

1. Value C will only be on in (Set A + Set B) if C is on in either A or B
2. Value C will only be on in (Set A - Set B) if C is on in A but not in B
3. Value C will only be on in (Set A * Set B} if C is on in both A and B

Ahh, get the idea? Worked in the same way you expected it to, didn't it? Well, in case you're still confused, here's a nice little example to illustrate this wonderful concept:

```
begin
  aSet := [1, 3, 5, 7];       { 1 3 5 7 }
  aSet := aSet + [2, 3, 5];   { 1 2 3 5 7 }
  aSet := aSet - [3, 5];      { 1 2 7 }
  aSet := aSet * [7];         { 7 }
end.
```

Yes! Yes! That's the way sets should be. Now that you know how to assign values to sets, it is time to take the next step, learning how to test if a value is on! (Hey, you did turn them on for something, right?)

The operator to use when testing if a value is on in a set is the *in* operator. For example, to test if the number 3 is turned on in aSet, you would type the following piece of code:

```
begin
  if (3 in aSet) then
    Writeln('Omigosh! 3 is on!');
end.
```

Rem - Now, we realise that you might be confused about why the operator is called 'in' - why isn't it called 'on', 'check', or something like that? Well, it's all very simple. The analogy we used for sets is to use a bunch of Check-Boxes. However, Pascal uses the analogy of *testing whether a value is inside the set or not*. (Now you know why it's called a set!) A set just contains a list list of values and *in* tests if the value you gave it is *in* that list. Confused? Well, don't worry, you don't need to know this to use sets - what you need is just a lot of common sense!

In the example above, the line 'Omigosh! 3 is on!' displays only when 3 is 'turned on' inside set aSet. Funny way of doing things, right? But never mind, once you get the hang of it, it gets better. And now that you've learnt the fundamentals of sets, we also think it's time for the example program! Here it is:

TYPE

```
program SetItUp;

  type
    TSet = set of 1..10;

  var
    i: integer;
    Numbers: TSet;

  begin
    Numbers := [3, 6, 8..10];
    for i := 1 to 10 do
      if (i in Numbers) then
        Writeln(i);
  end.
```

RUN

```
3
6
8
9
10
```

ANALYSE

Simple program, simple explanation. In fact, this program probably needs no explanation at all! But still, let's be fair, okay?

- ***Line 4*** *- This line defines type TSet which is a set of 1 to 10*
- ***Line 7*** *- This declares a counter variable*
- ***Line 8*** *- The set, Numbers, is declared here*
- ***Line 11*** *- Here, we assign for Numbers to have values 3, 6, 8, 9, 10 turned on*
- ***Line 12*** *- A for loop starts looping the counter from 1 to 10*
- ***Line 13 & 14*** *- Here, we test if the counter is turned on in Numbers. If it is, it executes Line 14, which prints out the counter. (You see how it works now?)*

End of Day 8

Ahh, now that we have finished sets, we can safely close this chapter on User-Defined Types... Or can we? Remember, many other different types are around, so your task isn't finished yet! But for now, since we can't continue anymore (without critical damage to our hands), we shall close this tutorial, to go on to *Day 9* - Pointers and Such!

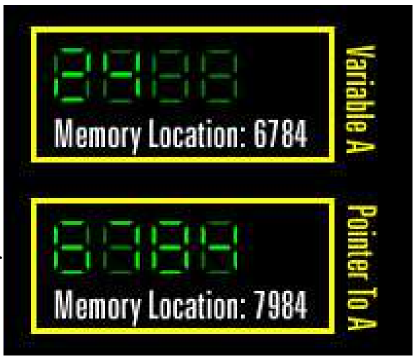
Class I
Day 9 - Pointers and Such

What you are going to learn today:

- What memory and pointers are
- How do you implement the mind-boggling pointers
- And of course, the example program!

We all need memory, right? People always give this advice: Get your computer more memory! Buy memory while it's still cheap! Use your old memory chips as samples to bring to class! Well today, we're not only going to introduce you to memory and pointers - two very dangerous topics (if wrongly dealth with, of course), but also to the concept of dynamic variables. Nothing difficult, so good luck!

What are pointers?



Em, remember our analogy of having boxes to represent variables back at Day 3? Well, let's go back to that analogy to understand pointers. Now, imagine that everytime you create a box (declaring a variable), it is located at a certain position in this huge shelf called 'memory'. This position is represented by a number. Now, what happens if we want to make another variable to trace the position and value of this example variable? Ahh, we use pointers!
Pointers, my friend, are simple variables which store the location of another variable. But this alone gives us almost unlimited flexibility. You may ask: "What the hell do I want with a variable location?!" Well, by obtaining the location of a variable, you can find the current value of that variable, manipulate the variable's value, create dynamic variables, and crash your computer into the great void.

Stop! - Wait! That last part of my sentence was totally true! If you carelessly declare and use pointers, they can actually cause your computer to crash - and crash badly. Why? Well, when you first create a variable, what it contains is trash - some crap left behind by some other program. Only when you give it a value is that variable 'safe' to use. The same is with pointers, except that the results are *far* worse. While undeclared variables normally just return 'trash' values, undeclared pointers, when used, may access memory locations which are sensitive to the computer's inner health, causing it to crash. So, remember to give values to pointers before you use them. Otherwise, you'll face the undesirable consequences!

Well, now that you know what pointers are, we can safely tell you what dynamic variables are all about too. Simply put, dynamic variables are variables you create while the program is running. Unlike variables created with the 'var' keyword, dynamic variables are created only when needed - which means that it has no variable name, just a place to store a variable. It's something like a last-minute box that is shoved in a position - nobody bothers to stick a label on something *that* last-minute. So, if it has no name, how do you get its value and do operations with it? Well, did you get the answer? Yes! You use a pointer to point to that memory location so that you can access it even without a variable name. Smart boy! Now, let's get on to talk about implementation...

Pointer Implementation

To implement pointers, you need to know two very sacred characters in Pascal: the caret (^) and the little-snail (@). To declare a pointer, first insert a caret in front of the data type, like this:

```
  type
    TPointer: ^integer;
```

Those lines declare a pointer to an integer. Of course, at this very moment the pointer is pointing to the great void, so we can't exectly do much with it, (unless crashing computers is your hobby). So, this is where the @ character comes in. That funny symbol returns the memory location of any variable, which you can assign the pointer to. See?

```
var
  MyPoint: TPointer;
  a: integer;

begin
  MyPoint := @a;
end.
```

So, in this example, MyPoint would be pointing to variable a because we had just passed it a's memory location. Now, this location isn't exactly very interesting - what we want is the *value being stored at this location.* So, to access this value, you use the caret again - but this time attached to the end of the variable name:

```
begin
  a := 24;
  MyPoint := @a;
  Writeln(MyPoint^);
end;
```

See that? MyPoint now has a caret attached when you want to access the value where it's pointing to. Amazing! But this isn't the only way to use a pointer. You can carve out a shelf space and place a value there at runtime. A dynamic variable! As described before, dynamic variables don't have a name, they rely on a pointer to manipulate its value. There are two commands to implement a dynamic variable: 'new' and 'dispose'.

To use new, just pass a name of a pointer, immediately space is allocated and the pointer is automatically set to point at that memory location. It's *that* simple. Now, what about dispose? Well, if you carve out space, you have to fill it back, right? To do this, use the dispose command and pass it the pointer. Every dynamic variable you create has to be disposed of - otherwise you'll experience a 'memory leak' where memory seems to get less and less, even though the memory in your computer stays the same.

- For those who are myopic or easily confused, let us reiterate this again: A pointer name, with no characters attached or anything, returns the memory location. A pointer name with a caret attached to the back, returns the value stored at the memory location. Don't be confused about which is which. In this case, you're actually passing the memory location for the procedure to create and delete, so the pointer name should be without the caret.

Dynamic variables are very powerful features, and you'll be amazed by what they can do. Because there is no limit to how many dynamic variables you can make (except your memory, of course), they can be used when you want to store values where the length is unknown. Want to see the new and dispose commands at work? Here goes:

```
begin
  new(MyPoint);
  MyPoint^ := 24;
  Writeln(MyPoint^);
  dispose(MyPoint);
end.
```

See that? We created a dynamic variable, gave it the value 24, printed it, and then disposed it permanently. Isn't that fun? Pointers, dynamic variables, memory, these terms aren't that scary anymore, aren't they?

Today's Example Program

Next up guys, is the last and final example program for Class I. Stretch your fingers - you're going to do some typing. Ahh, here we go:

TYPE

```
program PointyPoint;

  type
```

```
4:     TMyPoint = ^integer;
5:
6:   var
7:     Number, Number2: TMyPoint;
8:     i: integer;
9:
10:  begin
11:    Number := @i;
12:    i := 24;
13:    Writeln(Number^);
14:
15:    new(Number2);
16:    Number2^ := 24;
17:    Writeln(Number2^);
18:    dispose(Number2);
19:  end.
```

RUN

```
24
24
```

ANALYSE

This program is very simple, and as you'll see, it's got nothing to do with some technical mumbo-jumbo and dancing on the soil, so let's continue.

- ***Line 4*** *- Here we defind a new pointer type*
- ***Line 7*** *- We declare two pointers for use*
- ***Line 8*** *- We declare a counter*
- ***Line 11*** *- Now, the magic begins. We make Number point at variable i, so that if we access the value pointed by Number, it will return the value of i.*
- ***Line 12*** *- We now declare i to be 24*
- ***Line 13*** *- We use Number to print out i*
- ***Line 15*** *- A new dynamic variable is created*
- ***Line 16*** *- We asssign 24 to the dynamic variable pointed to by Number2*
- ***Line 17*** *- We print it out*
- ***Line 18*** *- And this make sure Number2 gets disposed of, and it's all over!*

End Of Day 9

This is the end of Class I, and well, it's been taxing. Loosen up, get a drink, go on a holiday. Ahh.. I can see it now... A lonely island in the middle of the Pacific Ocean. With the seagulls, the trees, the sand, the boat... Hey! Where's the boat?! Arrrgh! But never mind. Rest well, and we'll see you back at Class A, okay? (This was brought to you by Know-Better Travel Agencies.)

Class A

Welcome to Class A, and congratulations for making it this far! In case you're worrying about the difficulty level, nothing to fear, Class A is more of an introduction than an in-depth discussion of these nerve-wrecking topics. The code samples for all the example programs are also available, so go forth, and conquer the world without any worries!

***Day 10* - Introduction to Delphi**
Following the latest trends, Pascal has finally found its way *visually* into the Windows world. Battling Microsoft Visual Basic and winning, Delphi is probably the best choice if you're planning to develop Windows applications with blazingly fast program speed!

***Day 11* - Windows Programming**
Windows is new, but moving from DOS programming to full-blown Windows is very time consuming. So, Delphi is the perfect in-between development system for all of you. To help you make the migration easier, here's a short introduction to Windows with Delphi.

***Day 12* - Introduction to OOP**
You want hype? Well, the world of OOP is full of hype. Here we'll uncover the mysteries of OOP with Delphi, probably the most object orientated development system alive. This is an extremely short intro which just scratches the surface - there's still a lot more to learn...

***Day 13* - The Final Word**
The last and final tutorial for Class A is just a whole lot of introductions to topics like graphics, file I/O, and so on. Quick, easy, and short, don't expect in-depth discussions here! Flip through, enjoy, and learn something new. You just might need it in the near future!

Class A
Day 10 - Introduction to Delphi

What you are going to learn today:

- What is Borland Delphi
- How do you use Delphi
- And how we are going exploit it

You're not in the '90s if you don't know Windows - and the latest rage in programming are the Rapid Application Development (RAD) systems. RAD is just a term used to describe programming systems which make programming fast, easy, and efficient, and as you'll see later, this is totally true.

Delphi? Wealthy? Healthy?

Those of you who program in Windows may have heard of VB, the (in)famous RAD development system for Windows created by Microsoft. Although easy to use, VB has the following weaknesses:

1. VB programs are slow.
2. VB's interface is non-productive
3. VB does not allow you much flexibility
4. VB uses BASIC as its core language
5. VB is by Microsoft

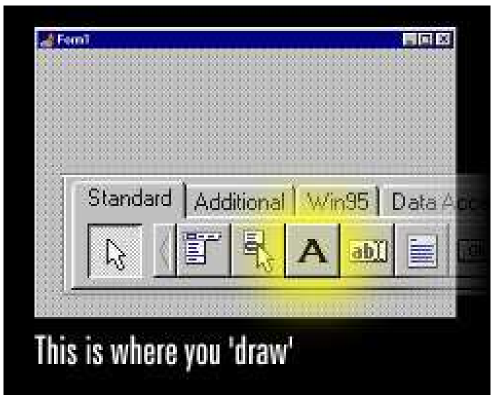
Delphi, on the other hand, is the exact opposite of all those points. Delphi, like VB, allows you to 'draw' your windows on your screen, then add code as you need it. Hmmm, very interesting...

Stop! - Ahhh, before you start thinking that Windows programming is the easiest thing in the world, I would like to tell you a story of my first real experience programming in Windows. It was a few years back, when I had just got a copy of Turbo C++ for Windows. I thought I could just type a few lines and make a great program, but believe it or not, even till today, I still can't make a functional Windows program with that package. So, thank your lucky stars that somebody thought of intelligent programs like Delphi. Just imagine typing hounds of code for one simple file dialog box!

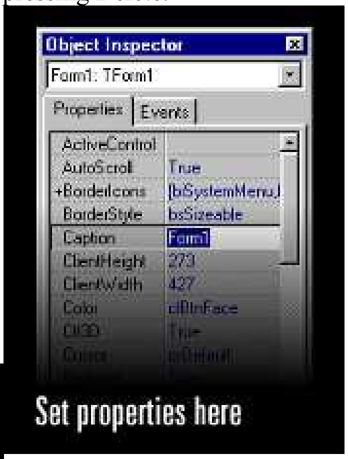
Essentially, there are two versions of Delphi: Delphi 1.0 and Delphi 2.0. But wait! This is just in - there's a Delphi 3.0! Essentially, Delphi 1.0 is for 16-bit development while Delphi 2.0 and 3.0 are for 32-bit development. Delphi 3.0 has features for developing ActiveX (just some more fun stuff from Microsoft), but because it's just too recent, we can't cover it in these tutorials.

How do we use Delphi?

First thing you must learn about Delphi programming - the process goes Draw, Change, Code. First, 'draw' your program on the screen, change what you've just 'drawn', then write code to accompany the program. To 'draw', you must first locate the big blank window on your screen which looks as if it had 'a bad case of measles'. (That line was lifted from Delphi Programming for Dummies!) This is where you start drawing your programs - select the button with the big 'A' on the top of the screen and drag across the 'canvas'. Instantly, a giant label appears on the screen, with the text: 'Label1'. Get the idea? Flip to the other tabs and see what other things you can 'draw'. Some things, however, don't show up on the screen. They just remain a small icon stuck there with

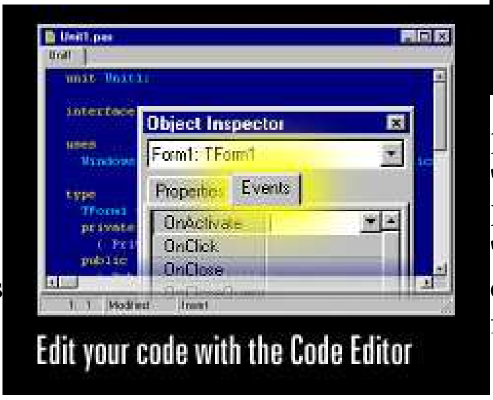


super-glue or something. Like any other painting tool, you can remove these items by selecting it and pressing Delete.
Now that you've drawn something, you might want to change their properties. For instance, you might want this Label to be blue. Simple, just use the Object Inspector to change it. In a sense, *you're changing what you've just 'drawn'*. Isn't that great? Much better than your old Crayola crayons.
To change an object property, you look it up on the Object Inspector, then change its value on the right. For instance, to change the text on the label, you can change its 'Caption' value to 'Delphi!' Some properties, you may also notice, have a '+' sign right before its name. This means that that property can be expanded by double clicking on its name - for instance, you can double-click on 'Font' and change its 'Color'. You'll see a pull-down list with all the possible values you can use to define its colour. Change it to clBlue for instance, or clGreen. You'll see that the colour of the text changes accordingly. Now you're getting the idea! Play around - All great fun! You're playing with colours, right? You can also double double click on the value to bring up a colour selection box. Ooo, ahhh...



ActiveControl	
AutoScroll	True
+BorderIcons	[biSystemMenu,]
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	True
Cursor	crDefault

Enough. Now that you have customised what you have 'drawn', the next step is to add code to each object. Look back at your Object Inspector. Notice that you've only been looking at the 'Properties' tab, the one you used to change your 'drawing'? Click on the second tab 'Events', and voila! You see the list of events linked to the object that is currently selected. See, the 'OnClick' event is activated when the object currently selected is clicked. Considering that you haven't done anything stupid, the following steps would create a 'Hello World!' application.



1. Click on the *form*, and go to the Object Inspector
2. Click on the Events tab
3. In the OnClick event, you can either type your own procedure name in the value column or just double-click to let Delphi handle the naming complications (We highly recommend the second option!)
4. Instantly, the Code Editor comes out for you to edit your code. Notice I said 'edit'? Well, that's because there's already code in your window! Believe it or not, Delphi created all that for you...
5. Type the following code between the 'begin' and 'end' blocks:
6. `Form1.Caption := 'Hello World!';`
7. Click on the green 'Play' button at the top of your screen

Form1.Caption
Object Name . Object Member

Wait for a moment as your hard disk lights flash... Poof! Instantly, a great window appears on your screen, beckoning you to click on it. Do so. At once, the title bar changes to say 'Hello World!' It works! Your first Delphi program works! Let's look at your code: First of all, as your code was for the form's OnClick event, your code would only be executed when the form gets clicked. When it does, it sets the *Caption* member of Form1, which is the form, to become 'Hello World!' Huh? Well, look at it this way. The form is actually a record, and you're setting the Caption property to become what you want it to be. (And as you'll see later, the form is not only a record, it's an object!) Just treat the form as a record for the moment and you'll be okay.

Exploitation Time!

Now that you've had a taste of Delphi, it's time to tell you of our devious plans... We plan to use Delphi to teach all of you the concepts of Windows and Object Orientated Programming, the two main topics of Class A. What? You don't have Delphi? Well, get it! Better now than never. Let's give you our top 10 reasons why you should get Delphi now:

1. We're all going to use Delphi in the future, anyway.
2. Delphi has all the source code you could ever want in a development system.
3. Delphi can get you results - fast!
4. Wait a minute! Delphi was fast in the first place!
5. Pascal is my sort of language, right?
6. Delphi comes with a fantastic Visual Component Library (VCL) which, em, never mind...
7. Delphi was a holy city in ancient Greece.
8. We all love Anders Hejlsberg!
9. Delphi is the love of the entire nation!
10. Borland Delphi is not Microsoft Delphi!

Now, wasn't that persuasive? Choke out your money, if you can, and go get your copy of Delphi at your nearest dealer! (Em, I'm still wondering who's the dealer for our area. Hmmm...) Whatever it is, just get it! Period.

End Of Day 10

As you can see, Delphi is very different from the console Turbo Pascal you've been learning so far. Who knows, you just might give up Turbo Pascal programming altogether! Some of you may be puzzled though. Why is it that Delphi executes code only when an event occurs? Well, you'll find out in Day 11! So good luck!

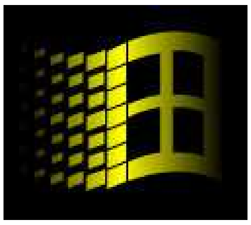
Class A
Day 11 - Windows Programming

What you are going to learn today:

- What is Windows
- How do you Program in Windows
- How Delphi works in Windows

Windows Programming is what Delphi is about, but to use it efficiently, you must know a little bit of Windows' "Base System Architechture" (just some funny term thought up by some idiot...) Flip through and absorb - Windows Programming is one of the most interesting topics around, for us that is.

What is Windows?



Skipping its infamous history, you can say that Windows is just simply the result of envious PC owners who wished that their computer would be 'easier to use'. When Apple Computer released the Macintosh, (which I still hail as the greatest invention of all time) developers started working on a similar operating system for the PC. Through some marketing and lots of tension, Windows became the forerunner of the whole lot. Today, Windows has become a sort of standard for all PCs. Almost all new computers are pre-installed with Windows 95, (I personally prefer doing the installation myself, but never mind), and almost all Windows programs today are comparable, or even better, than their DOS counterparts. How about that for a success story?

How do you program in Windows?

Now that you know more about the history of Windows, we shall now talk about how Windows programming works. This chapter will make it easier for you to understand Delphi. (You are getting confused, right? Right?) You'll find out that Windows programming isn't that difficult - after you understand it of course.

First thing about Windows - it is a *multi-tasking environment.* That means that it actually caan run more than one program at one time, (supposedly) giving you higher efficiency. That gave the creators of Windows a lot of headaches. How can you have multiple programs running at the same time? You'll need to process code for each program simultaneously, which I assure you, is no easy feat. So, the makers of Windows created a system where each program was like a procedure - code was run only when the system requested for it. This meant that they needed to create a whole new system for Windows, containing the API (Application Program Interface) and other forms of trash. Code was in the form of 'events'. Only when an event occurs will code be run. Now you know why Delphi makes you type code that way!

But that's not all. Remember I told about something called the API? (It's just up there, stupid!) The API is actually just a whole lot of commands built into Windows to do what programmers need for Windows Programming. You may not know it, but Delphi gives you access to the API too! But don't worry if you look up the API and find millions of cammands - I bet you'll never use 85% of them in the end anyway.

Another thing. Have you ever wondered how come I like to start type definitions with the letter 'T'? Well, we call this Hungarian Notation and it's just a naming method to help programmers identify the data type of a variable quickly. I mention this because some of you may be getting the misconception that the 'T' is compulsary, while in actual fact the naming method comes from me programming in Windows too much. How come? Well, suffice it to say that Windows uses Hungarian Notation a lot, so look out! When you start typing SDK code, you may not recognize if it's a typo or if it's real!

How Delphi implements Windows

As you've already seen, Delphi implements Windows very closely to how Windows programming is supposed to work. You attach code to events and API calls are allowed in your code. Delphi is probably the most efficient Windows programming environment isn't it? But that's not all. Tomorrow, you'll learn about Object-Orientated Programming, and you'll see how efficient Delphi is after all.

Class A
Day 12 - Introduction to OOP

What you are going to learn today:

- What is Object-Orientated Programming
- How do you use Object-Orientated Programming
- And how Delphi is Object-Orientated

You've heard the hype, now learn the real thing. As you'll soon find out, Objects are just extremely functional ways of organising information. However, don't expect to learn everything. After all, you still want your sanity after this, right? Topics like polymorphism and virtual functions don't even get a single sentence here!

What is Object Orientated Programming?

Like all other programmers, I shall use the analogy of a watch to explain the concept of objects. A watch is like a record, isn't it? It has values like the current date, the temperature. and of course, the time. In a record declaration:

```
type
  TWatch = record of
    date: TDate;
    temperature: TTemp;
    time: TTime;
  end;
```

Now, a watch not only stores values, it also has procedures and functions built inside them. For example, each time you press a button, a procedure is run to change the display. Or maybe the procedure changes the time. We don't know. But you've got the idea don't you? An object is just a lot of things put together into one. OOP scientists like to call this *encapsulation.*

But what makes an object so special? Well, look at it this way. Imagine that you're going to make a new watch which is exactly the same as the old one except that it can show the compass directions. (Yes, there are such watches around.) Instead of re-typing everything, what objects allow you to do is to *inherit* from already existing objects. That is, to make a new object from an existing one ane modify it. Useful, isn't it?

Another feature of abjects is its ability to *protect* variables. This feature prevents procedures from modifying values they shouldn't be touching. For example, the watch wearer may want to change how long a second is. However, because you have protected it, the user has no way to modify the length of the second. On the other hand, if the watch, for some unearthly reason, wants to shorten the length of a second, it can be done - that is if you allow them to of course.

How is OOP implemented in Pascal?

Well, we originally planned for Delphi to be the 'teaching tool' for this chapter, but implementing it gave some problems. So, instead we'll be using Turbo Pascal instead. Delphi's OOP implementation has also changed from Turbo Pascal's, so don't expect the code to be portable!

But never mind - to program OOP in Pascal, you must first create a class. A class, which represents the watch in our analogy, can be inherited into another class. A class also encapsulates other things, like how Class A encapsulates Days 10, 11, 12, and 13. Want to see how a class definition looks? Well, here it is:

```
  type
    TThing = object
      public
        i, j: integer;
    end;
```
But that's only the simplest implementation for a class (also known as an object) - in fact, not only can you place values inside classes, you can also place procedures and functions! Here's an example:
```
  type
    TThing = object
      public
        procedure WatchMeCry;
    end;

  procedure TThing.WatchMeCry;
    begin
      Writeln('Waaahhh!');
    end;
```
Huh? Don't worry if you don't understand it. We'll explain it to you bit by bit. Let's look at the first example first. Like a record, it stores two fields - i and j. See it's structure is just like a record? But wait! What's that 'public' doing there? Well, it's there to define what type of 'visibility' the fields after it will be. The different keywords are:

1. *Public* makes the declarations after it become available to all other classes. That means that a class totally unrelated can still access that value if it is public,

2. *Protected* makes the declarations available only to the current class and descendant classes. (You'll learn about inheritence later)

3. *Private* makes only the value accessible only by the class itself.

Here, object members, i and j, are declared public, making them accessable through any class. Yes, any other class. However, if you had made them private, you would not be able to access them through other classes. But now another question arises. How do you refer to these member values?
```
  begin
    Thing.i := 7;
    Thing.j := 1;
  end.
```
Yes, you use the same dot operator you used with records. See the similarities now? An object is just like a record in almost every way. Ahh, good news for us. We can skip over 40% of this introductory stuff now...

But how about our second example? About declaring procedures and functions? Actually, it's very simple. When you're declaring the class, just add a procedure statement which is the same as the *procedure statement you type when declaring the procedure*. For example:
```
  type
    TThing = object
      public
        procedure WatchMeCry(Length: integer);
    end;
```
That line is just there tell Pascal that procedures exists - but at the moment it'a undefined. Now, how do we give the procedure some code to run? Simple, we would have to declare it later in the code with *the name of the class*, the dot operator, then *the name of the procedure*. Just like this:
```
  procedure TThing.WatchMeCry(Length: integer);

    var
      i: integer;

    begin
      for i := 1 to Length do
```

```
        Writeln('Waaahhh!');
    end;
```
See the name of the procedure? Now you get what I mean. The method to implement functions is the same too. So, you can actually call up a function just like this: (Magic, actually, magic...)
```
  var
    Thing: TThing;

  begin
    { Prints 'Waaahhh!' 5 times }
    Thing.WatchMeCry(5);
  end.
```
Bet you've never seen anything like that before! OOP is pure power, but hold on - it's not even starting yet! (If it was that simple, we would have put it with the part on records instead of wasting so much time typing.) Objects are more than what you've just seen.

For instance, can you inherit a record? Well, for classes, the answer is yes. To inherit a class from another, place the ancestor class between brackets after the keyword 'Object'. See this?
```
  type
    TGone = object(TThing)
      public
        Zoo: integer;
    end;
```
What have we just done? Well, we have just created a new type TGone which is the same as type TThing except that it adds a new field - Zoo. Remember what are the aims of inheriting? It's not just for adding new functions, you know. We can also modify existing ones:
```
  type
    TGone = object(TThing)
      public
        procedure WatchMeCry;
    end;

  procedure TGone.WatchMeCry;
    begin
      Writeln('Crying Out Loud!');
    end;
```
As you can see, as long as you make a new method which has the same name as an existing one on an ancestor class, it will completely replace it - including the parameter list. Quite powerful, eh? Now let's have a concluding example program to round it all off, shall we?

TYPE

```
program ObjectFolly;

  type
    TThing = object
      public
        Money: integer;
        procedure KillMoney;
    end;

  var
    MoneyThing: TThing;

  procedure TThing.KillMoney;

    begin
```

```
16:      Money := Money - 10;
17:    end;
18:
19:  begin
20:    MoneyThing.Money := 100;
21:    MoneyThing.KillMoney;
22:    Writeln(MoneyThing.Money);
23:  end.
```

RUN

```
90
```

ANALYSE

This program is generally a simple example of OOP, there's still a lot to learn, so look out! I mean, look in the help if neccesary...

- ***Line 4** - Starts off object TThing*
- ***Line 5** - This makes things public*
- ***Line 6** - An example variable*
- ***Line 7** - An example procedure*
- ***Line 11** - Declares a variable of type TThing*
- ***Line 13, 15, 16, 17** - Defines the procedure and makes it minus 10 from object field Money.*
- ***Line 20** - Makes MoneyThing.Money to be 100*
- ***Line 21** - Run procedure in MoneyThing to minus 10 from Money*
- ***Line 22** - Prints everything out, and ends the program.*

Oh, that was fun. Now that you've got the basics, we can ask this very simple question: How is Delphi Object-Orientated? Well, the answer's right below.

How is Delphi OOP?

When you first start up Delphi, do you notice that there's something which looks like this at the top of the code listing?

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Believe it or not, that's an object! And if you know Delphi well enough, you'll know that everything in Delphi, a label, a button, anything, is an object by itself! Everything! Utterly everything!

- You might have noticed that in place of the keyword 'object', you had 'class'. Well, as we've said before Delphi did change some aspects of their OOP model. So, you'll probably have to look up the help to upgrade your skills. (We can't teach everything, you know.)

The End Of Day 12

Yes, we've covered OOP, and now it's your last day. Ahhh, this is going to be sad. But hey, you're gonna leave anyway, so why not finish the course first? Day 13 is there for you - Go on and take that final step! Bye!

Class A
Day 13 - The Final Word

What you are going to learn today:

- Simple Shape Graphics
- The Write command
- How to continue learning Pascal

Today's the last day - and it just so happens to be unlucky number 13. But don't fret, today's tutorial is short and sweet. We'll first introduce you to graphics, file I/O, and finally a 'Where to go from here' section. It's been nice teaching you - Good luck.

Shapey Shapes With BGI

Don't you have a powerful pc? I mean, you have SuperVGA with that great high-rez monitor which can cram pixels up to 1024 by 768 right? So, why don't you make use of it? Pascal allows you to fully utilze your pc's graphic utilities with the Graph unit, so read on!

The first thing you must do when writing a graphics program is to initalize the graphics. You do this with the InitGraph procedure. You need to pass it three parameters, these are: first, the value of which graphics driver to use, second, the graphics mode it's in, and third, the Borland Graphics Interface (BGI) files path, or the directory where the BGI files are found. (Most likely it's C:\TP\BGI, but hey, we don't know which directory you installed Pascal to, so we can't exactly be perfect.) Note however, that you have to pass *variables* to the procedure, not just values, because InitGraph refers to its parameters by reference - It actually changes the values of the integers. For example:

```
uses
  Graph;

var
  driverVar, modeVar: integer;

begin
  driverVar := Detect;
  Writeln(driverVar, ' ', modeVar);
  InitGraph(driverVar, modeVar, 'C:\TP\BGI');
  Writeln(driverVar, ' ', modeVar);
  Readln;  { To Pause The Display }
end.
```

If you run the above program, you'll notice that the values of driverVar and modeVar are different before and after InitGraph. Normally, variables you pass won't be changed by a procedure because what it does is to just copy the value. But here the original value is changed instead. You can implement this in your procedures too - just add the var keyword before the variable names you want to be referenced.

But enough of that. You're probably more intrigued by the code listing above. Let's start slowly, shall we? After we include the Graph unit, we go on to declare driverVar and modeVar - both integers. On the very first line, we make driverVar to be equal to Detect. Detect? What Detect? Well, Detect is just a constant representing 0 declared in unit Graph. Thus the following lines would all be the same:

```
driverVar := Detect;
```

```
  driverVar := 0;
  driverVar := Detect + 0;
  driverVar := Detect - 0;
  driverVar := a + b + c + d - a - b - c - d;
```

Great! Now we print out the values of driverVar and modeVar - Use InitGraph on them, and print them out again. The last line Readln is just there to pause the display so that you won't need to go to the User Screen to see it all again. Whew! After we initialize the graphics, we can now start drawing!

1. **Drawing Boxes**

For the drawing of boxes and other forms of rectangles, you would use the Bar command. You pass it four values - first two are the XY coordinates of the *top- left corner*, last two are the XY coordinates of the *bottom-right corner* of the rectangle. Thus, to draw a 10 by 10 square, you'll type:

```
  Bar(10, 10, 20, 20);
```

Now that was a great square, eh? But what if you want it to fill the whole screen? Well, there are the GetMaxX and GetMaxY commands, which return the screen width and screen height in pixels. So, this is how it'll look:

```
  Bar(0, 0, GetMaxX, GetMaxY);
```

Well, you get the idea now. It's all very simple, actually. Now, after rectangles, you have - of course, circles! (Little Round Things...)

2. **Drawing Circles**

Circles are generally as simple as rectangles, but you need a bit more knowledge of maths. You pass the Circle command three parameters. The first two are the X and Y coordinates of the *center of the circle*. The third is the *radius*, which is the distance between the center and the edge. So, try this out:

```
  circle(10, 10, 10);
```

This line will draw a circle with the center at coordinates (10, 10), extending 10 pixels from the center. Nice effect, eh? But what if you want to draw an imperfect circle, otherwise known as an ellipse? Well, you'll use the ellipse command:

```
  ellipse(0, 0, GetMaxX, GetMaxY);
```

That draws a nice big ellipse all across your screen - huge one. Now is that powerful or what? Now, after the program finishes, you've still got one last thing to do. Return the computer back to text mode. Use the command CloseGraph to do this. It isn't neccesary, but highly recommended...

3. **Drawing Lines**

Lines can be drawn using the line command. It accepts four parameters: first two are the XY coordinates of the *start of the line*, the next two are the XY coordinates of the *end of the line*. So it looks like this:

```
  Line(12, 15, 80, 90);
```

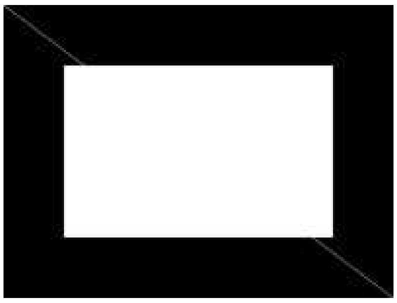
Ahh, this draws a line from (12, 15) to (80, 90). Great work! Who said that graphics were a difficult topic? We just finished this whole thing in one chapter!

Well, now it's time to embark on your typing spree. Go on. Type this program. If you dare. (Um, we would like to state that we are not responsible for anything stupid which happens during this program...) And yeah, you can change the BGI path if it is different from what is stated on the listing. Here goes:

TYPE

```
program DrawTheHeck;

  uses
    Graph;

  var
    driverVar, modeVar: integer;

  begin
    driverVar := Detect;
    InitGraph(driverVar, modeVar, 'C:\TP\BGI');
    Line(0, 0, GetMaxX, GetMaxY);
    Bar(100, 100, GetMaxX - 100, GetMaxY - 100);
    Readln;  { This is to pause the display. }
    CloseGraph;
  end.
```

RUN



ANALYSE

This is a very simple graphics program which outputs a stupid looking picture, but hey, it works! So who's complaining? (The video card, perhaps?)

- ***Line 4*** *- This loads unit Graph*
- ***Line 7*** *- This declares the two variables*
- ***Line 10*** *- Makes InitGraph auto-detect by setting ModeVar*
- ***Line 11*** *- Initializes Graphics*
- ***Line 12*** *- Draws a line from the top left corner of screen (0, 0) to the lower right corner (GetMaxX, GetMaxY)*
- ***Line 13*** *- Draws a box starting (10, 10) and ending (GetMaxX - 10, GetMaxY - 10)*
- ***Line 14*** *- Readln freezes the display until key is pressed*
- ***Line 15*** *- This returns the screen back to text mode and the program is over...*

The Write Command

As you know, the Writeln command prints a line and then moves the cursor to the next line, but what if you don't want that. Well, you use the Write command, which keep the cursor at where the line ended. Thus, the following code...

```
begin
  Write('This is ');
  Write('my life!');
end.
```

...would output the following:

```
This is my life!
```

As simple as that, you've joined two separate statements together! Wow! That is powerful, isn't it? (Actually, this is pretty basic knowledge, but never mind.)

Where to go from here...

From here, we have finished everything we have on our agenda. Well, here is the list of things we never taught you about, but you should definitely try learning:

1. File I/O
2. Writing units
3. Advanced OOP
4. Advanced Delphi and Controlling interrupts

Well, it's been long and hard, but now you can certainly say you were 'enlightened'. After all, this was helpful, wasn't it? What to do now? Well, maybe you could go back and clarify some of your thoughts before you leave this place permanently - or, maybe you could just browse around. Who knows? You just might learn something new. See you around, programmer!

Almost Complete Reference

Welcome to the Almost Complete Reference (ACR)! This section is supposed to be used as a reference for the tutorials, but you can come in and look around too. (Just make sure you don't go overboard and print out the entire section) The topics are all arranged in alphabetical order, so as to make searching easy. Happy Reading!

A B C D E F G H I J K L M N O P R S T U V W

ActiveX Controls

ActiveX, created by Microsoft, is a fun and well, not so fun, control which can be used on Internet pages. Like Java, it has the ability to make Internet pages interactive and fun, but can take quite a bit of effort to write. Find out more from the Microsoft site at www.microsoft.com.

Array Types

Arrays are essentially a data types which simulate tables - one-dimensional, two-dimensional, whatever. Arrays are extremely useful when you need to transfer a lot of information at one go.

Bar Command

The Bar command draws a bar on the screen. It accepts four parameters - the co-ordinates of the top-left corner and those of the bottom-right corner. Whoo, now that's scary...

BASIC Language

BASIC stands for Beginner's All-Symbolic Instruction Code - a no-brainer programming language used for simple programs. BASIC generally has a simpler syntax, but the price is that your code generally becomes almost unreadable. It was only when Microsoft revamped the language and gave it some flair did people really start to pay attention. Historical note: In the old days, almost all programs were written in BASIC because it was the only language around then - it must have been difficult!

Bill Gates

For those who have been gone from the planet for the last century or so, Bill Gates is the CEO of Microsoft - and the richest man in America. (Don't confuse him with Bill Clinton, that's the President of the United States...) Microsoft was co-started by him, and well, look where it is today. Oh, I wonder how he does that...

Borland International

Borland International was, and still is, the best company dealing with language compilers in the world today. Although it also publishes other programs, Borland is most well known for its compilers, and was also the creator of Turbo Pascal, the compiler which popularised Pascal as a programming language.

The Case Statement

The case statement is used to replace massive numbers of if statements. By comparing a value to a list of constants, case decides which code is run, which is not. Simple as that.

Circle Command

The circle command accepts three parameters, two for the coordinates, one for the radius. (No, you don't need a degree to understand this, just Primary 6 knowledge.)

C Language

The predecessor to C++, this slightly less powerful language is still widely used today. Originally created by At&T, it has become one of the most important languages in the world. (For programming, that is.) People say that the only reason why C hasn't died out is because C programmers felt that C++ was too powerful for their needs. Wait a minute - is there any logic here?

C++ Language

Like Pascal, C++ is another language programmers use to create programs. Although similar to Pascal, C++ is slightly harder to learn, but is signifigantly more powerful. (Notice that there's always a price to pay for everything?) As mentioned above, it seems that the reason why C++ isn't taking over the world is because people felt that C++ was too powerful for their needs... huh?

Const Keyword

The const keyword is used to declare constants - values which never change and can be represented by somrthing else. Just by declaring constants in the const section, you will be able to use that term thoughout the entire program. Great fun! Constants also help make code easier to read and debug by replacing meaningless values with meaningful names. That's what constants are for.

Dispose Command

The dispose command is used to remove space created by the new command. By providing it with the pointer name, the dynamic space will be removed instantly. Notice that you don't have to worry about getting rid of normal variables - that's done automatically.

Ellipse Command

The ellipse command accepts four parameters like the Bar command, except that it draws an oval. Ooo, isn't that beautiful?

The For Statement

The for statement makes a counter loop upwards or downwards from a certain specified number to another. This feature makes it possible to repeat a portion of code and have avariable which changes on each turn. Very useful - but you'll have to think of course.

GetMaxX, GetMaxY Commands

GetMaxX is the command which passes the maximum X co-ordinate for the screen and GetMaxY does the same for the Y co-ordinate. Ahh, power in Graphics...

Hungarian Notation

Hungarian Notation is the use of prefixes, and sometimes suffixes, to identify a variable's data type. This method of naming is used excessively in Windows, which makes it absolutely neccesary to learn it. Thankfully, with Delphi, you don't have to care a bit, but a little knowledge helps...

The If-Else Statement

The If-Else statement is the simplest example of decision-based control flow. It tests a condition, and if it is true, a block of code will execute. If it is not, you can either specify it to do nothing, or make it run another block of code. As we've said before, it is simple...

InitGraph Command

InitGraph is the command which initializes the graphics drivers for your computer. You pass it three variables, one for the driver, another for the mode, and the third which is the path where the BGI files are stored. Yes, that's how you do it...

Line Command

The line command draws a line from the four parameters you passed it. The first two are the coordinates for the start, the last two for the end. Yes, yes! Lines! Bwa-ha-ha-ha! (Sleep Deprivation Syndrome)

Macintosh

The Macintosh was a computer created by Apple which became famous for its robustness and user-friendliness. Remembered for its achievements in computer design, it helped prevent the PC from going into the voids of techno gibberish. Even till today, a lot of people use the Macintosh in their daily lives, and yes, Macintoshes are also known as 'Macs' - something that has lived on through the years...

Memory

Memory - the greatest thing the world has ever known. Somehow, every creature is stuck on trying to get more memory. Thankfully, I found the cure: Don't buy Windows games.

New Command

The new command is used to make dynamic space for a new dynamic variable. It accepts a pointer as a parameter and declares space according to the pointer's data type. Oh well, what could be easier?

Null-Terminated String

Also known as a C style string, this is probably the most complex thing you've seen so far. Pascal offers null-terminated strings with the 'PChar' type - but be forewarned, they're not so easy to work with!

Object-Orientated Programming

The new wave of programming has begun with the new concept of OOP, which deals with things called objects, data types very similar to records. Now, OOP is a new sort of programming method which allows you to inherit and override procedures. Confused? Well, you don't expect us to be able to explain this all in one short paragraph, right? You'll have better luck with Day 12 - Introduction to OOP, so shoo!

Parameters
To pass a value over to another procedure or function, use a parameter. The procedure will then get the value and process it. Note that the value is copied. Any changes will not affect the actual value.
Pascal Language
Pascal is a programming language first made famous by Borland's Turbo Pascal compilers. Today, although not so widely used, it's a good thing to have Pascal somewhere in your resume if you're applying for a job which involves programming. The signifigance of Pascal has also grown since Borland released Delphi, the 16 and 32 bit programming environment for Windows. So, Pascal's not that bad a choice, eh?

Rapid Application Development
Rapid Application Development systems are systems which forego the tiresome and error-prone process of Windows programming, and makes it easier to produce 'quality' programs quickly - however, the first generation of RAD development systems were pretty disappointing because of their program speed. Thank god for Delphi! Without it, god knows where RAD systems would be today!
Record Types
Records are data types which can store other values together. For instance, a student record can hold five different values representing the student's test scores. This is really useful when you need to organise information.
The Repeat-Until Statement
The repeat-until statement repeats a certain bit of code until a condition is met. Thus, the condition must be false for the loop to continue. A very close cousin to this statement is the while statement.

Set Types
Set types are just like checkboxes - they allow values to be turned on and off. For instance, you could have a set defining the properties of a character. If it's Bold and Italic, the Bold and Italic values will be turned on. Yup. It's as simple as that...

Turbo Pascal
Turbo Pascal is the name of the famous compiler which sold millions when it was first released by Borland. Today it still exists - and we'll be using it for this page. You can find out more about it at www.borland.com, or visit one of the many websites dedicated to distributing Pascal programs like this one. There's a lot of information out there!

Uses Keyword
The uses keyword is used to load units, little files which define new procedures, functions, variables, and constants for your program. Something like the concept of a plug-in. (You know what I mean, right?)

Var Keyword
The var keyword is used to declare variables and to specify their data types. For instance, in the var section, you could declare a variable i which is an integer. As simple as that. Hey, who said programming was difficult?
Visual Basic
Visual Basic was the development system which first started the excitement over Rapid Application Development. Created by Microsoft, it probably has the largest user base because of its immense popularity and the easy-to-learn nature of the language. (And yes people, I started Windows programming this way!)

The While Statement
The while statement repeats a block of code while a condition is still true. Unlike the repeat-until statement, the while statement guarantees that the code will be run at least once. Who knows? You just might need it someday, right?
Windows
Microsoft Windows is the 'operating system' (for Windows 3.x) which was created for the PC to make it emulate the Mac. For some reason, it became 'the standard' for all PCs. Well, we'll never understand it - so let's live with it, okay?
Write Command
Write is the command to use when you want to print a line of text without breaks. Like the Writeln command, it can also accept variables, which make this quite an invaluable command...
Writeln Command
Writeln, which is a close cousin of the Write command, prints a line out, and then moves the cursor to the next line. This is a standard command which lets you output text to the screen. If you want to prevent the cursor from going to the next line, use the Write command instead.